Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática

A Case-Based Approach to Software Design

Paulo Jorge de Sousa Gomes

Coimbra
December 2003

Thesis submitted to the
Universidade de Coimbra
for the partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Informatics Engineering
Speciality of Artificial Intelligence

A Case-Based Approach to Software Design

Paulo Jorge de Sousa Gomes

Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática
ISBN 972-9039-69-0
December 2003

Thesis under supervision of
Doutor Carlos Lisboa Bento
Professor Auxiliar do Departamento de Engenharia Informática
da Faculdade de Ciências e Tecnologia
da Universidade de Coimbra

Para os meus Pais.

Para a Ana, por todos os momentos felizes que passámos juntos.

Abstract

The growth in software complexity is one of the main reasons for the increase of errors in software, along with a clear raise of the development time. One way to overcome this problem is the development of CASE[1] tools capable of helping the software engineer in a more intelligent and powerful way. One possible approach for such a tool, is to reuse software development knowledge, which enables faster development and increased software quality. This approach can be complemented with knowledge management methodologies that operate at corporate level, thus beneficiating all the software engineers in the corporation. The work presented in this thesis describes a software design reuse system based on Case-Based Reasoning (CBR).

Our approach complements an UML[2] CASE tool with a knowledge management tool for software design knowledge. We selected the design phase of software development, because decisions made at this phase are complex and have great impact in the implementation. The developed model is able to help the software designer by: retrieving similar class diagrams; suggesting alternative designs; proposing design changes based on design patterns; checking a diagram for errors and correcting them; helping the knowledge base administrator in the case base maintenance; and using natural language for software object naming.

This thesis describes our approach and its implementation (the REBUILDER system). Several experiments were made and the results are reported, along with an extensive study on related systems, positioning REBUILDER. This system is a first step to the development of a commercial CASE tool that addresses the support of software design and design knowledge management. One of the main goals was to test several reasoning mechanisms in the reuse of UML class diagrams, demonstrating that this approach is feasible and suitable for a CASE tool.

[1] Computer Aided Software Engineering.
[2] Unified Modelling Language.

Acknowledgements[3] / Agradecimentos

Esta tese representa o culminar de mais uma etapa na minha carreira científica. Apesar de ter sido um trabalho que me deu imenso prazer a fazer, o caminho percorrido até aqui não foi um passeio, mas sim uma prova de meio-fundo com o passo acelerado. Para este documento ter sido concluído a contribuição de algumas pessoas foi preponderante, às quais quero aqui prestar os meus agradecimentos.

Todo este trabalho não teria sido possível terminar sem o apoio dos meus pais, aos quais eu quero agradecer o que me ensinaram. A confiança que sempre tiveram em mim, bem como o respeito das minhas decisões, foram factores cruciais para ultrapassar os problemas e barreiras deste desafio. Um grande bem haja para eles.

Apesar da minha motivação para levar este trabalho a bom porto, foi um trabalho muito exigente e por vezes esgotante. É aqui que o apoio e a presença da Ana foram fundamentais. Não quero deixar de lhe agradecer por todos os momentos de felicidade que passámos juntos, que contribuiram para voltar ao trabalho revitalizado e motivado. Obrigado Ana, que sejas muito feliz.

Ao Professor Carlos Bento quero agradecer todo o seu apoio e confiança nas minhas decisões. Sendo ele o grande responsável por eu ter enveredado por estes caminhos, não quero deixar de ressalvar que este trabalho não poderia ter sido concluído sem o seu empenho. Quero agradecer também a sua amizade e o previlégio que me concedeu de o ter acompanhado em vários desafios. Um grande obrigado Professor, e que os nossos próximos desafios sejam tão ou mais motivantes do que este.

Esta tese foi um trabalho de equipa, sem a qual não se poderiam ter atingido os objectivos do projecto REBUILDER. Aqui quero agradecer ao Professor José Luís Ferreira por ter sido o responsável administrativo do projecto e por toda a disponibilidade que sempre teve. Aqui quero destacar o empenho que teve durante o painel de avaliação do projecto em Lisboa, aonde se 'bateu' com a Dra. Manuela Veloso com um claro resultado positivo, como se verificou com a aprovação do projecto. Ao

[3] **Due to personal reasons, this section is written in Portuguese. My apologies to the Portuguese non-speaking readers.**

Francisco C. Pereira quero agradecer a sua amizade, e todo o trabalho que teve a fazer de 'advogado do diabo' aquando das reuniões do projecto. Sem as suas críticas, comentários e conhecimentos o trabalho teria sido sem dúvida mais pobre. Quero agradeçer ao Paulo Paiva pelo ano e meio de trabalho que teve no projecto, bem como pela sua dedicação e capacidade de resposta às exigências de implementação do projecto. Ao Nuno Seco quero agradecer o seu empenho e a eficiência com que abraçou algumas das tarefas do projecto. Ao Paulo Carreiro agradeço também a sua dedicação e a tenacidade com que ultrapassou diversos problemas relacionados com o projecto. Quero agradecer à Paula Mano e à Manuela Carrão pelo apoio contabilístico ao projecto. Para todos eles os meus votos de um futuro muito positivo.

Ao Professor Amílcar Cardoso agradeço o seu apoio administrativo e a sua amizade durante estes anos de trabalho. Aos meus amigos e colegas do Grupo de Inteligência Artificial (*aka* AI Lab), agradeço o seu apoio, companhia, amizade e opiniões, que foram importantes para o estabelecimento de um óptimo ambiente de amizade e de trabalho. Em particular gostaria de agradecer ao Penousal Machado, Francisco Baptista Pereira, Leonor Melo, Carlos Grilo, Dr. Ernesto, Miguel Ferrand, Luís Macedo, Pedro Gago, André Dias, Jorge Tavares, Eduardo Taborda e Ricardo Ferrão. Queria agradecer também ao Paulo Marques pela disponibilização do *cluster* sem as quais não teriamos conseguido efectuar os testes de desambiguação.

Uma das actividades que indirectamente mais contribuiu para a realização deste empreendimento foi a natação e os meus companheiros de piscina, com os quais eu habitualmente passava as minhas horas de almoço[4]. Para a Sofia, o Ricardo, o Pedro, o Jorge, o Pedro 'Farfala', a Filipa, o Nuno, a Andreia, a Maristela, o Dani, o Paulo, o Ricardo Esperança, o Toni, o Mauro e para os Masters da AAC, um grande Obrigado e que continuemos a nadar e a retirar desta actividade um grande prazer.

À minha família uma palavra de apreço e de agradecimento pela sua ajuda e apoio quando necessário, com especial destaque para a minha avó sempre com um coração grande e generoso. Um grande abraço para todos eles.

O trabalho de investigação descrito nesta tese foi desenvolvido no Laboratório de Inteligência Artificial da Universidade de Coimbra.

Este trabalho foi financiado pelo POSI - Programa Operacional Sociedade de Informação da Fundação Portuguesa para a Ciência e Tecnologia e pela União Europeia através do programa FEDER, sob o contracto POSI/33399/SRI/2000, e pelo programa PRAXIS XXI.

[4]Isto até que meia dúzia de cabeças pensantes tivessem a brilhante ideia de mandar demolir as piscinas sem haver contrapartidas ...

Contents

List of Tables

List of Figures

Resumo Alargado em Português

Este capítulo tem como objectivo dar uma ideia global e resumida do trabalho desenvolvido no âmbito desta tese. A primeira secção descreve quais as motivações, objectivos e contribuições deste trabalho. Esta secção faz também uma introdução aos principais conceitos utilizados neste trabalho. Em seguida descreve-se a arquitectura do sistema desenvolvido (o REBUILDER), bem como os vários módulos que o compõem. A terceira secção deste capítulo descreve o trabalho experimental efectuado, apresentando os principais resultados obtidos com os vários módulos do sistema. Seguidamente é feita uma descrição geral dos principais trabalhos que têm relação com o trabalho desta tese. São descritos vários tipos de sistemas, sendo feita uma comparação das principais características desses sistemas com o REBUILDER. A finalizar são apresentadas as contribuições desta tese, trabalho futuro e direcções de investigação a prosseguir.

Introdução

Nesta secção são descritas as motivações deste trabalho, bem como os conceitos básicos presentes nesta tese. Seguem-se os objectivos e contribuições deste trabalho.

Motivações e Enquadramento

O desenvolvimento de software é uma tarefa cada vez mais dispendiosa em termos de recursos. A dimensão e complexidade dos sistemas de informação desenvolvidos pelas empresas de software aumenta de dia para dia. Este facto faz com que as empresas de desenvolvimento de software tenham cada vez mais dificuldade em entregar o software dentro dos prazos estipulados pelos clientes. Muitas das vezes, a qualidade dos sistemas produzidos é negligenciada de forma a se poder fazer a entrega de uma versão a tempo. Com vista a solucionar ou atenuar este problema, a comunidade

científica da área de engenharia de software propôs, à cerca de uma década, como caminho possível a reutilização de software [Prieto-Diaz, 1993, Coulange, 1997].

No início, a reutilização de software incidia sobre o código, através da reutilização de rotinas e/ou funções que estavam em repositórios de software ou em bibliotecas de código [Burton et al., 1987, Prieto-Diaz and Freeman, 1987]. Uma das razões que levou a que inicialmente a reutilização incidisse sobre o código, foi a ideia de que (entre outras possíveis razões) reutilizar código é 'fácil' porque este está formalizado. No entanto, existem outros níveis de reutilização do conhecimento resultante do processo de desenvolvimento de software, desde o nível da especificação de um sistema, até ao nível dos testes, passando pelo *design*[5]. Em última análise, qualquer tipo de conhecimento presente no processo de desenvolvimento de software pode ser reutilizado.

Os conhecimentos adquiridos pelos elementos das equipas de desenvolvimento é um bem inestimável para uma empresa de software. Este conhecimento ou experiência, designado em Inglês por *know-how*, permite que os seus portadores reutilizem todo um conjunto de conhecimento adquirido no desenvolvimento de software, construindo novos projectos mais rapidamente e com maior qualidade.

O abandono de colaboradores experientes, leva a que as empresas percam o *know-how* dessa pessoa, o que geralmente constitui uma quebra em termos de produtividade. Num mercado global em que a competição é muito grande, este tipo de perdas leva consequentemente a uma perda de competitividade por parte da empresa. Uma forma de resolver este problema (ou pelo menos atenuá-lo) seria através do recurso a um repositório de conhecimento capaz de armazenar e gerir parte do *know-how* dos vários colaboradores, ou seja a empresa ter um sistema de gestão de conhecimento [Liebowitz, 1999]. Outra vantagem de ter um repositório de conhecimento, seria que este poderia ser partilhado com os vários colaboradores da empresa, de forma a haver uma reutilização deste conhecimento de uma forma mais produtiva.

Dentro das diversas fases de desenvolvimento de software (análise, *design*, implementação, testes e integração, assumindo o modelo em V de desenvolvimento de software [Boehm, 1988]), a fase de *design* é pouco abordada na reutilização em comparação com a fase de implementação. Isto é em parte explicado pela falta de uma formalização das especificações de *design*, demonstrada pela falta de uniformização

[5]Vamos utilizar a palavra em Inglês pois não nos foi possível encontrar uma palavra em Português que pudesse transmitir o conceito que está inerente à palavra em Inglês.

ao nível de linguagens de especificação do *design*[6]. Outro possível factor é o grau mais elevado de abstracção do *design* em relação ao código, o que torna a reutilização mais complicada, tendo que ser tratada a um nível de abstracção mais alto aonde a subjectividade é também um obstáculo. Outra dificuldade que explica a menor reutilização de conhecimento da fase de *design*, é a própria natureza do processo de *design*. Não existe uma teoria forte quanto ao *design* de um sistema de software. Existem várias metodologias, no entanto, estas apenas apresentam linhas de orientação para situações abstractas, deixando para o *designer* (engenheiro de software ou responsável pelo desenvolvimento do *design* de um sistema) a sua implementação para situações específicas. O sucesso de um bom *design* para um sistema de software depende mais da experiência dos *designers* do que da metodologia usada, isto porque o conhecimento sobre o *design* de software ainda não está suficientemente desenvolvido e formalizado para dar lugar a uma engenharia mais exacta. Neste aspecto voltamos a frisar que o *know-how* do *designer* é fundamental para o sucesso de um *design*.

Alternativas recentes sobre a reutilização de especificações de *design* andam à volta de *frameworks*[7] [Johnson, 1997], padrões de *design* [Gamma et al., 1995] ou componentes [Atkinson et al., 2002]. No caso das *frameworks*, a reutilização é feita através de uma reinstanciação de um conjunto de classes que formam a estrutura básica e o núcleo de um determinado tipo de aplicação. O trabalho do *designer*, está em ampliar e configurar este conjunto de classes de forma a adaptá-las ao caso específico que está a ser desenvolvido. Um padrão de *design* consiste numa solução para um problema abstracto de *design*, ou seja, um padrão de *design* descreve como se devem organizar e projectar um conjunto de classes e interfaces de forma a solucionar um problema típico de *design* de software. O desenvolvimento de software recorrendo a componentes consiste na utilização de componentes de software (classes ou conjuntos de classes) de forma a construir um sistema ou parte dele. Na utilização de componentes, a ideia é reutilizá-los como se fossem blocos básicos de construção de software. Estas abordagens são distintas entre si, no entanto, elas têm em comum a possibilidade de reutilização do *design* juntamente com o código. No caso dos componentes e das *frameworks* esta intenção é mais clara, onde o que se reutiliza em termos de *design* é o *design* da *framework* ou dos componentes. Nos padrões, isto já não se passa bem

[6]Pelo menos no início da prática da reutilização de software. Hoje em dia linguagens como o UML (Unified Modelling Language, [Rumbaugh et al., 1998]) vieram uniformizar estes formalismos.

[7]Palavra que designa uma estrutura ou infra-estrutura de software, mas que, no entanto, decidimos que não deveria ser traduzida, pois iria desvirtuar o conceito expresso na palavra em Inglês.

assim, neste caso a reutilização é feita a um nível mais abstracto, que pode não ter que ver necessariamente com o código. Este nível mais alto de abstração foi uma das razões que nos levou a utilizar padrões de *design* na nossa abordagem. Desta forma os padrões de *design* são integrados no nosso sistema como uma ferramenta de *design*.

Como referimos anteriormente, o aumento da dimensão e complexidade do software, levam a que a fase de *design* do desenvolvimento de software ganhe cada vez mais importância. Isto porque os erros cometidos ao nível do *design* vão ter fortes consequências ao nível da implementação. A importância do *design* faz também com que as equipas de desenvolvimento sejam mais eficientes e mais criativas nos *designs* que desenvolvem. Novas maneiras de projectar os sistemas de software são necessárias de forma a se poder obter uma optimização do tempo de desenvolvimento, bem como aumentar o desempenho do sistema que está a ser construído. Como foi dito anteriormente, o processo de *design* baseia-se muito na experiência dos *designers* responsáveis. Na maioria das áreas de engenharia mais amadurecidas a reutilização de componentes é uma técnica bem conhecida. No caso da engenharia de software a reutilização de componentes não é simples, e ainda há um longo caminho a percorrer nesta matéria. Uma das formas de se poder auxiliar o *designer* a reutilizar componentes ou qualquer outro tipo de conhecimento resultante do desenvolvimento de software, consiste na utilização de ferramentas CASE[8] capazes de reutilizar este conhecimento. Estas ferramentas teriam que expandir as já existentes de forma a juntarem capacidades cognitivas mais complexas. A estas ferramentas vamos chamar ferramentas ICASE (ferramentas CASE Inteligentes, *Intelligent CASE tools* em Inglês). Para além da capacidade básica de reutilizar *designs* anteriores em novas situações, estas teriam de ir mais além, fornecendo funcionalidades que envolvem raciocínios mais complexos, tais como a exploração de *designs* alternativos de forma a fomentar a criatividade dos *designers*, ou fazer uma identificação e correcção automática de erros sintácticos e semânticos que possam existir num *design*.

A criatividade [Boden, 1990] é um fenómeno importante para o *design* em geral, e para o *design* de software em particular. A criatividade no *design* pode ser vista de uma perspectiva mais funcional em que novos *designs* são desenvolvidos tendo um melhor desempenho do que *designs* anteriores, ou então de uma perspectiva mais estética em que, por exemplo, sejam menos complexos. Em termos cognitivos a criatividade

[8] *Computer-Aided Software Engineering*.

pode ser definida como um processo que dá origem a um produto dito criativo, com base na avaliação de certas características deste produto [Dasgupta, 1994]. Podem ser identificados quatro componentes distintos na criatividade [Brown, 1989]: o processo, o produto, a pessoa ou entidade criadora, e a situação em que se desenrolou o processo. Neste modelo o produto é crucial para a identificação da criatividade, isto porque associada à identificação da criatividade está geralmente o produto que é considerado criativo. O estudo do processo criativo é também importante para o desenvolvimento de modelos computacionais de criatividade. Trabalhos anteriores sobre *design* criativo identificaram pelo menos quatro métodos que geralmente se encontram associados à geração de *designs* considerados criativos [Gero, 1994b]: analogia, combinação de ideias, geração a partir de primeiros princípios e transformação do espaço conceptual de *design*. A entidade criadora de um produto considerado criativo geralmente refere-se a um humano, no entanto, julgamos que os sistemas computacionais possam ser considerados como uma entidade criadora. Este é um assunto polémico que é discutido no seio da comunidade científica que estuda a criatividade e cujo debate está fora do âmbito desta tese. O quarto aspecto a ter em conta é a situação ou contexto em que se desenrolou a criação do produto criativo, o qual tem um papel preponderante na classificação do produto como sendo criativo ou não. Este aspecto também não é abordado neste trabalho pois também está fora do âmbito estabelecido para esta tese.

Dentro do *design* podem ser definidas três classes de *design* [Gero, 1994a]: o *design* rotineiro, o *design* inovador e o *design* criativo. O *design* rotineiro acontece quando não existe a criação de novos *designs*, ou seja, o espaço de *designs* não é modificado. O design inovador consiste na geração de novos *designs* com base na utilização de novas gamas de valores para as variáveis de *design*, expandindo desta forma o espaço de *design*. O *design* criativo pode ser definido como o processo que gera novas classes de *designs*, alterando radicalmente o espaço de *designs* através da alteração do conjunto de variáveis de *design*. Os *designs* gerados possuem características que fazem com que sejam designados como criativos, estas propriedades definem linhas de orientação para o avaliador, sendo igualmente importantes para o estudo do *design* criativo e consequentemente para o desenvolvimento de abordagens computacionais ao *design* criativo. Do nosso ponto de vista existem duas características principais: originalidade e utilidade. Originalidade porque o *design* criado tem que ser substancialmente

diferente dos já existentes. Utilidade, pois tem que ter as funcionalidades desejadas pelo criador, existindo aqui uma intenção associada ao processo de *design* criativo.

Muitos dos sistemas de reutilização de software (ver capítulo 6 para mais informação sobre estes sistemas) apenas fornecem ajuda na recolha e selecção de componentes de software, como classes, funções ou especificações, de um repositório central. No entanto, na maior parte das vezes, os componentes seleccionados necessitam de modificações de forma a serem adaptados à situação corrente. Esta tarefa de modificação de componentes é uma tarefa mais exigente do ponto de vista cognitivo devido à sua complexidade e à quantidade de conhecimento que geralmente é necessário utilizar para efectuar as modificações. O raciocínio analógico [Gentner, 1983, Hall, 1989, Holyoak and Thagard, 1989] é muitas das vezes utilizado como uma forma de adaptar uma solução de um outro problema a uma situação diferente. Este funciona através do mapeamento estrutural e semântico de características das duas situações seguido da transferência da situação fonte para a situação alvo. Este processo não só permite a adaptação de soluções, como também permite a criação de novas soluções, eventualmente com um potencial criativo.

Neste trabalho uma das apostas que é feita, é na utilização de um mecanismo de raciocínio analógico para gerar novas soluções de forma a que o *designer* possa explorar outras regiões do espaço de *design*, estimulando o potencial criativo do *designer*.

As ferramentas CASE foram desenvolvidas de forma a auxiliar os engenheiros de software nas tarefas de análise e *design* de um sistema de software, existindo actualmente alguns sistemas CASE que geram as declarações das classes, interfaces, métodos e atributos. Estas ferramentas têm como principais funções automatizarem a verificação sintáctica dos diagramas criados, para além de serem editores de diagramas. Um sistema CASE é geralmente composto por vários módulos: editores de diagramas, editores de texto, repositório de informação, bem como módulos mais específicos. A maior parte destes módulos, como os editores, é funcionalmente simples e fornece ao *designer* um grau elevado de auxílio. No entanto, existe um módulo que tem sido pouco explorado - o repositório de informação. Vários tipos de conhecimento podem ser armazenados no repositório de informação de forma a poderem ser úteis aos *designers*. Ferramentas que sejam capazes de reutilizar o conteúdo do repositório de informação são necessárias para efectuarem esta tarefa. Acresce ainda, que se o repositório é extenso, o que é comum, este torna-se quase impossível de explorar pelo

designer e só com o recurso a uma ferramenta computacional é que é viável. É aqui que algumas das técnicas de Inteligência Artificial podem dar um grande contributo, transformando os sistemas CASE em ICASE. As ferramentas ICASE seriam capazes de armazenar e reutilizar conhecimento resultante do desenvolvimento de software de uma forma inteligente, pondo ao dispor dos *designers* um conjunto de mecanismos cognitivos úteis. Alguns destes mecanismos são, entre outros: sugestão de soluções alternativas, exploração do repositório de conhecimento guiado pela semântica, verificação semântica de diagramas, aprendizagem da interacção com os *designers*.

A linguagem UML (Unified Modelling Language, [Rumbaugh et al., 1998]) é uma linguagem de especificação e modelização de software que utiliza vários tipos de diagramas, de forma a conseguir modelizar diferentes aspectos de um sistema de software. O UML é uma das linguagens de especificação de software mais usadas em todo o mundo (senão a mais usada). Devido a este sucesso e porque é um excelente meio de comunicação entre analistas, *designers* e programadores, a linguagem UML é seguida neste trabalho como base de representação de *designs*. De todos os tipos de diagramas definidos no UML, seleccionámos os diagramas de classes para serem utilizados como principal fonte de conhecimento. Estes diagramas descrevem o sistema modelizado de uma forma estática, descrevendo os objectos envolvidos bem como atributos e métodos. Estes diagramas são dos mais usados dentro do UML, daí a escolha deste tipo de diagramas no nosso trabalho.

Quando um *designer* experiente começa a desenvolver o modelo de um sistema de software, este não começa do zero. Os *designers* usam sempre conhecimento de projectos anteriores sob a forma de experiência. O raciocínio baseado em experiência é uma forma muito comum de raciocínio humano, tendo sido usado para inspirar uma forma de raciocínio em Inteligência Artificial: o Raciocínio Baseado em Casos (RBC, [Kolodner, 1993]). O RBC funciona com base em experiências representadas na forma de casos, que constituem os blocos básicos de conhecimento utilizados na resolução de novas situações com semelhanças relativamente aos casos em memória.

O RBC pode ser visto como uma metodologia de desenvolvimento de sistemas baseados em conhecimento [Althoff, 2001] que raciocinam com base em experiência. A ideia principal do RBC é a de reutilizar conhecimento passado de forma a resolver problemas actuais.

Geralmente um caso contém três partes: problema, solução e resultado. O problema descreve a situação que o caso representa. A solução descreve o que é que foi usado para solucionar o respectivo problema. E o resultado descreve o desfecho da aplicação da solução ao respectivo problema. Os casos são armazenados e indexados num repositório chamado de biblioteca de casos ou base de casos.

A um nível abstracto o RBC pode ser descrito por um ciclo de raciocínio com as seguintes fases [Aamodt and Plaza, 1994] (ver figura 1) : recolha, reutilização ou adaptação, verificação ou revisão, e retenção ou aprendizagem[9]. No início de um ciclo de raciocínio tem que ser especificado um problema alvo, que consiste na descrição do problema que se pretende resolver. A primeira fase consiste na recolha de casos da biblioteca de casos com base na relevância em relação ao problema alvo. A relevância de um caso é na grande maioria das vezes definida com base na semelhança do caso com o problema alvo. A fase de adaptação, se tiver lugar, tem como objectivo a modificação da solução do melhor ou melhores casos recolhidos relativamente ao problema alvo. O resultado será uma nova solução, que juntamente com o problema alvo constituem um novo caso. A fase seguinte é a verificação da nova solução, que permite identificar e resolver erros e inconsistências nessa solução. Desta fase pode resultar que a solução não consegue solucionar o problema e o sistema volta à fase de adaptação, escolhendo um outro caso recolhido de forma a ser utilizado para a geração de uma nova solução. Finalmente, a fase de retenção pode armazenar o novo caso na base de casos, dependendo esse armazenamento dos critérios de aprendizagem do sistema. Fecha-se assim o ciclo de RBC, permitindo ao sistema aprender novas experiências.

No início deste trabalho a ideia central era a de construir um sistema capaz de ajudar o *designer* na reutilização de software e na gestão de um repositório de software ao nível da empresa. A primeira funcionalidade foi conseguida através de uma ferramenta CASE que utiliza UML e que coloca à disposição do *designer* um conjunto de funcionalidades cognitivas para reutilização de software baseadas em RBC, analogia, padrões de *design* e composição de *designs*. O segundo objectivo é conseguido usando o RBC como metodologia de gestão de conhecimento. O RBC foi a abordagem central na construção do sistema, agregando ainda um conjunto de técnicas que são descritas nesta tese.

[9]Em Inglês: *retrieve*, *reuse*, *revise* e *retain*.

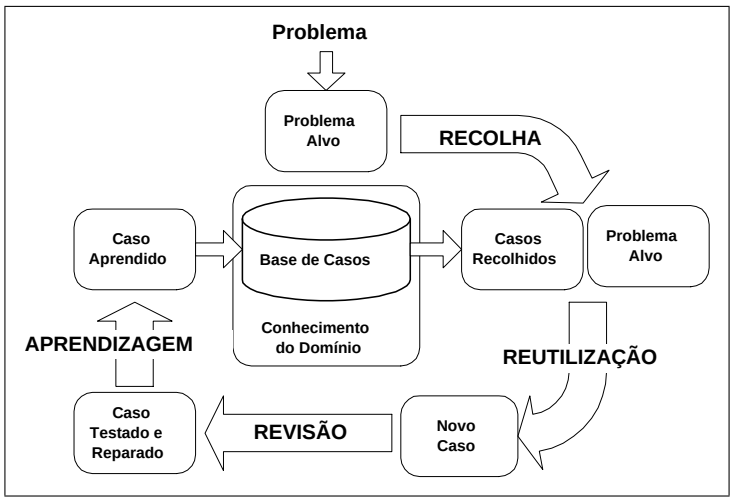


Figure 1: O ciclo de Raciocínio Baseado em Casos adaptado de [Aamodt and Plaza, 1994].

Objectivos e Contribuições

Tendo como ponto de partida a importância da fase de *design* no desenvolvimento de software e a necessidade de técnicas que permitam a um sistema a reutilização de *designs*, o objectivo principal deste trabalho é o desenvolvimento de um sistema capaz de reutilizar *designs*. Este sistema deverá ter duas características centrais: armazenar e gerir um repositório centralizado de *designs* de software, e fornecer aos *designers* mecanismos capazes de reutilizar estes *designs* de uma forma racional.

O REBUILDER tem uma arquitectura cliente-servidor, que se adequa bem a um ambiente distribuído como o de uma empresa de software. Isto vai de encontro à primeira característica considerada necessária no armazenamento centralizado, enquanto que a parte cliente do REBUILDER, que consiste numa ferramenta CASE que utiliza UML, permite fornecer aos *designers* meios para a reutilização racional dos *designs* disponíveis. Esta ferramenta funciona como um editor de UML normal com funcionalidades extra que permitem ao utilizador reutilizar *designs*.

Com base nos objectivos iniciais e com o decorrer do trabalho, novos assuntos e sub-objectivos foram aparecendo. Os objectivos e contribuições consideradas centrais a este trabalho são descritas seguidamente, enquadradas segundo as técnicas que foram aplicadas:

- O objectivo principal é a integração da vertente ICASE com a vertente de gestão de conhecimento de *design* de software. Aqui a contribuição deste trabalho é a arquitectura cliente/servidor desenvolvida, bem como a forma como se dividem

e interagem os vários módulos.

- A representação de casos é um dos pontos essenciais para o bom funcionamento dum sistema de RBC. Neste ponto optámos por uma representação de casos centrada no utilizador, ou seja, o sistema utiliza a 'linguagem' do *designer* de software (o UML) em vez de ser o oposto. Neste aspecto julgamos que este é um aspecto inovador deste trabalho.

- A indexação e a recolha de casos são dois factores importantes para a performance de um sistema de RBC. Um dos objectivos deste trabalho é o desenvolvimento de um esquema de indexação e recolha flexível e eficiente em termos de complexidade temporal. Neste ponto foram conseguidas duas contribuições importantes. Um esquema de indexação baseado no WordNet em que não só os casos são indexados, mas também os objectos dos casos, o que permite uma maior flexibilidade de recolha de casos. Outra contribuição são os algoritmos de recolha e as métricas de semelhança desenvolvidas.

- De forma a sugerir ao *designer* vários *designs* alternativos e ajudar na exploração dos espaços de *design*, a reutilização de casos foi abordada neste trabalho de uma forma alargada, tentando explorar diversos mecanismos de adaptação. O resultado foi o desenvolvimento de três formas diferentes de adaptar diagramas de casos antigos a um novo problema:

  - Adaptação de diagramas de classe através de raciocínio analógico. Existem várias contribuições neste ponto, nomeadamente a forma como o WordNet é utilizado para selecionar os objectos candidatos a mapeamento, a forma como os mapeamentos são efectudados e o trabalho de experimentação desenvolvido sobre o estudo do *design* criativo.

  - Outro tipo de adaptação desenvolvido, que utiliza vários casos para solucionar um problema e que foi por nós designado por composição de *designs*.

  - A utilização de padrões de *design* de software[10] para a modificação e evolução de *designs*. Esta contribuição é uma das que entendemos como mais originais pois é a primeira abordagem que automatiza todo o processo de aplicação de padrões de *design*, desde a escolha do padrão até à

[10] *Software Design Patterns* em Inglês.

aplicação do padrão escolhido.

- A fase de verificação permite a filtragem de erros e incoerências que resultam da fase de adaptação. Sendo um dos principais objectivos deste trabalho a modelização do ciclo de RBC completo, a integração desta fase no modelo desenvolvido é essencial. Este aspecto foi abordado com base em casos de verificação que representam conhecimento sobre o domínio e que vão servir para identificarem erros em diagramas. Estes casos são utilizados também para fazer correções em diagramas. Para além dos casos de verficação, outra contribuição deste trabalho é o modo como cada *designer* tem a sua base de casos de verificação específica, o que permite a personalização deste mecanismo. Outro aspecto inovador é a utilização do WordNet para fazer verificação.

- Um sistema de RBC pode tirar vantagem da fase de retenção para adquirir mais conhecimento, nomeadamente novos casos, podendo assim melhorar a sua performance. A implementação de um módulo de aprendizagem é um dos objectivos deste trabalho, bem como os aspectos de manutenção da base de casos que lhe estão associados. Neste domínio este trabalho adapta várias estratégias de manutenção de casos para trabalharem com uma representação de casos baseada em grafos.

- Um dos objectivos principais deste trabalho é o de melhorar a interacção entre o *designer* e o sistema CASE, o que é feito através da utilização de processamento de linguagem natural. A utilização de linguagem natural no REBUILDER está limitada à representação de casos e em específico à atribuição de nomes aos objectos de software. Um dos problemas que é abordado neste trabalho é o de tentar compreender qual o conceito associado a um determinado objecto com base no nome deste, designada por desambiguação de palavras[11]. Neste campo, para além de desenvolvermos dois métodos de desambiguação foi feito um estudo alargado comparando vários métodos descritos na literatura.

A secção seguinte descreve o REBUILDER e os seus vários módulos.

[11] Em Inglês *word sense disambiguation*.

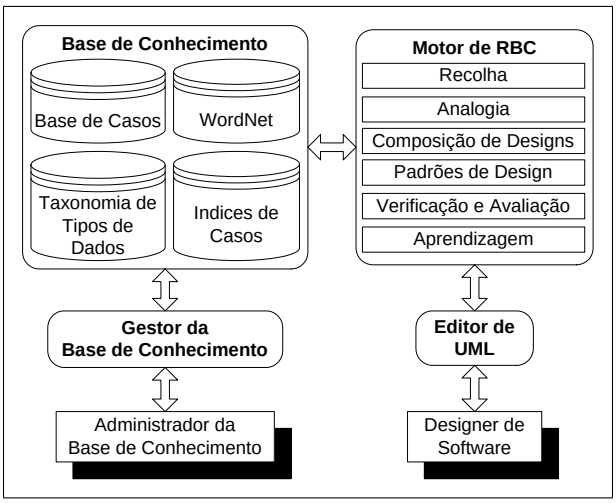


Figure 2: A arquitectura do REBUILDER.

REBUILDER

Esta secção descreve o modelo desenvolvido no âmbito deste trabalho e que é levado à prática através do REBUILDER. Começamos por descrever a arquitectura do sistema e mostrar como é que os diversos módulos interagem entre si. Seguidamente passamos a descrever cada um dos módulos do sistema (ver figura 2):

- Editor de UML.
- Base de conhecimento.
- Gestor da base de conhecimento.
- Motor de RBC.

Arquitectura do Sistema

O REBUILDER tem dois tipos de utilizadores: *designers* de software e administrador da base de conhecimento. Os *designers* utilizam o REBUILDER como uma ferramenta CASE com capacidades de reutilização de *designs*. O administrador da base de conhecimento tem por função gerir a base de conhecimento de forma a esta se manter actualizada e coerente. O editor de UML é o interface entre o sistema e o *designer*, enquanto que o gestor da base de conhecimento faz o interface entre o sistema e o administrador.

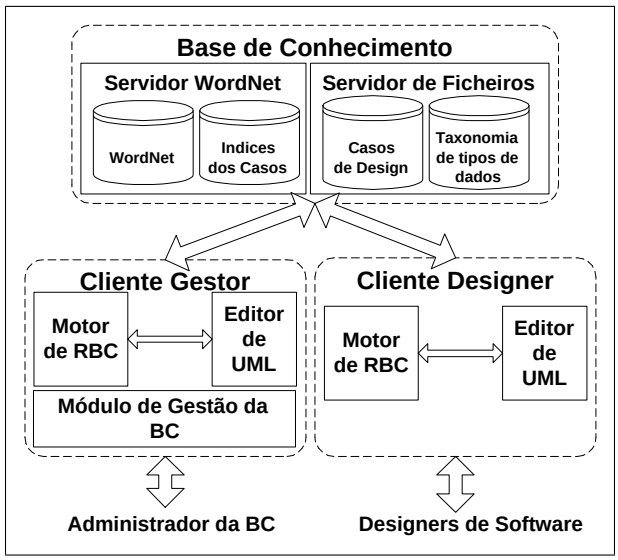


Figure 3: A arquitectura do REBUILDER de um ponto de vista de implementação.

O REBUILDER é baseado numa arquitectura cliente-servidor que engloba dois servidores e dois tipos de clientes (ver figura 3). A base de conhecimento (BC) é composta por dois servidores: O WordNet que contém o WordNet e os índices dos casos, e o servidor de ficheiros que comporta a base de casos e a taxonomia de tipos de dados. Enquanto que o primeiro servidor responde a pedidos sobre o WordNet e sobre os índices dos casos por parte dos clientes, o servidor de ficheiros trata de todos os pedidos referentes aos casos e à taxonomia de tipos de dados. Os dois tipos de clientes existentes são semelhantes entre si. Um é o *cliente gestor*, responsável pela manutenção da BC, é portanto o interface com o administrador da BC. O *cliente designer* tem como função ser a ferramenta ICASE do REBUILDER, a ser utilizada pelos *designers*. O *cliente gestor* tem todos os módulos existentes no *cliente designer* e acrescenta mais um, o módulo de gestão da BC. O *cliente designer* é constituído pelo editor de UML e pelo motor de RBC, responsável por todas as inferências do sistema. Apenas pode existir um servidor do WordNet, um servidor de ficheiros, e um *cliente gestor*. Em relação ao *cliente designer*, podem haver quantos clientes os recursos do sistema o permitirem, cada um correspondendo a um posto de trabalho.

O editor de UML para além de ser o interface do REBUILDER com os utilizadores, constitui também o ambiente de modelização para o *designer*. O editor integra os comandos de edição e de manipulação de objectos de UML habituais num editor de UML, e ainda os comandos de reutilização de *designs* que executam acções por parte

Table 1: Correspondência entre fases do ciclo de RBC e módulos do REBUILDER.

Fase do Ciclo de RBC	**Módulos do REBUILDER**
Recolha	Recolha
Adaptação	Analogia, Composição de *Designs* e Padrões de *Design*
Revisão	Verificação/Avaliação
Aprendizagem	Aprendizagem

do motor de RBC.

A BC é constituída por quatro elementos: base de casos, que armazena e organiza os casos; a taxonomia de tipos de dados, que é uma hierarquia de tipos de dados utilizados pelo sistema; os índices dos casos, usados para melhorar a performance do sistema; e o WordNet, que é utilizado como uma ontologia genérica.

O módulo de gestão da BC é usado pelo administrador. Este módulo contém todas as funções necessárias para a gestão da BC, que incluem a gestão da base de casos, dos índices dos casos, e da taxonomia de tipos de dados.

O motor de RBC é responsável por todo o raciocínio no REBUILDER. Como o nome indica, o RBC é o mecanismo central de raciocínio no REBUILDER que integra outros tipos de mecanismos para a fase de adaptação. Este módulo é composto por seis sub módulos: recolha, analogia, composição de *designs*, padrões de *design*, verificação/avaliação, e aprendizagem. Em termos de ciclo de RBC estes módulos distribuem-se da forma descrita na tabela 1.

Editor de UML

Como editor de UML foi usado o ArgoUML[12] que é software de utilização livre[13]. A construção de um editor de UML está fora do âmbito desta tese, de forma que foi escolhido um editor cujo código estivesse disponível para alteração. Os comandos do motor de RBC e do gestor da BC foram integrados no editor de forma a serem utilizados pelos *designers* ou pelo administrador. Estes comandos podem ser divididos em dois grandes grupos:

- **Comandos sobre a Base de Conhecimento:**

[12]A página web do ArgoUML é http://argouml.tigris.org/.
[13]Do Inglês *open source*.

  - Criar uma nova base de conhecimento.
  - Abrir uma base de conhecimento.
  - Fechar a base de conhecimento.
  - Invocar o gestor da base de casos.

- **Comandos sobre Acções Cognitivas:**
  - Recolher e seleccionar *designs* e objectos.
  - Reutilizar *designs* por analogia.
  - Reutilizar *designs* por composição de *designs*.
  - Reutilizar *designs* utilizando padrões de *design*.
  - Verificar *designs* e corrigir respectivos erros.
  - Avaliar *designs* utilizando métricas de engenharia de software.
  - Usar estratégias de manutenção de bases de casos.
  - Submeter *designs* à base de casos.
  - Gerir a classificação de objectos.
  - Modificar os parâmetros de configuração dos mecanismos de raciocínio.

Base de Conhecimento

A base de conhecimento (BC) contém o conhecimento necessário para os mecanismos de raciocínio. Esta é constituída pelas seguintes partes: base de casos, WordNet, índices de casos, e taxonomia de tipos de dados.

A base de casos tem dois tipos de casos: casos que representam *designs* (chamados de casos de *design*) e casos que representam aplicações de padrões de *design* (casos DPA - *Design Pattern Application*)[14]. Os casos são armazenados em ficheiros, um ficheiro por caso, e apenas são lidos para memória quando necessário. Cada caso tem um ciclo de vida, podendo estar em um de três estados: não confirmado, confirmado, e obsoleto. Um caso é considerado não confirmado quando é submetido à base de casos por um *designer*, e fica a aguardar que o administrador o examine e o declare útil ou inútil para a base de casos. Quando o caso é considerado útil passa para

[14]Em todo o documento, quando nos referirmos a casos, sem especificar qual o tipo de casos, estamos a referir-nos a casos de *design*.

confirmado. Só os casos neste estado é que podem ser utilizados pelos mecanismos de raciocínio. Um caso passa a obsoleto quando atinge o fim do seu ciclo de vida, seja porque ficou desactualizado, seja por qualquer outra razão. A base de casos possui três listas, uma para cada estado possível. Cada lista engloba os casos que estão no respectivo estado, por exemplo, a lista de casos confirmados contém todos os casos que são utilizados pelos mecanismos de raciocínio.

No REBUILDER um caso descreve um *design* de um sistema de software, que é representado através de diagramas de classe. Um caso é composto por: nome usado para identificar o caso; *package*[15] principal, que contém todos os outros objectos do caso; e nome do ficheiro que contém o caso. Os objectos de UML considerados no REBUILDER são: *packages*, classes, interfaces e relações.

Os casos DPA descrevem uma situação específica, aonde um padrão de *design* foi aplicado a um diagrama de classes. Cada caso DPA contém: descrição do problema e descrição da solução. O problema descreve em que situação foi o padrão de *design* aplicado (diagrama de classes inicial) e como foi aplicado (os participantes mapeados, ou seja, os objectos que têm um papel activo na aplicação do padrão). O diagrama de classes inicial é o diagrama aonde foi aplicado o padrão de *design*. Os participantes mapeados são objectos do diagrama que devem estar presentes na aplicação do padrão de *design*. Um participante pode ser um objecto, um atributo ou um método. Cada participante tem uma função específica dentro de um padrão de *design*, e cada padrão tem o seu conjunto específico de participantes. A partir do momento que os participantes estejam identificados num diagrama, a aplicação do padrão de *design* respectivo é feita de forma automática pelo mecanismo de aplicação de padrões de *design*. A solução de um caso DPA é o nome do padrão aplicado. Para mais detalhes sobre casos DPA ver a secção 3.3.1.

O WordNet[16] [Miller et al., 1990] é usado no REBUILDER como uma ontologia genérica. Na concepção do WordNet é usada uma teoria diferencial aonde os conceitos são representados por símbolos que permitem a distinção entre estes conceitos. Os símbolos são palavras, os conceitos são designados de *synsets*. Um *synset* é um conceito que pode ser representado por uma ou mais palavras. Palavras que podem ser usadas para representar o mesmo conceito são designadas de sinónimas, e uma

[15] *Package* é o nome em Inglês dado a um tipo de objectos dos diagramas de classes de UML que servem para agrupar elementos de UML.

[16] Mais informação sobre o WordNet pode ser obtida na página http://www.cogsci.princeton.edu/ wn/.

palavra com mais de um *synset* é considerada polissémica. O WordNet está construído à volta do conceito de *synset*. Basicamente contém uma lista de *synsets* cada um com as respectivas palavras, que podem ser usadas para o representar. Depois existe um conjunto de relações semânticas entre *synsets*. As relações são do tipo: *is-a*, *part-of*, *member-of*, *substance-of* e outras. Os *synsets* são classificados em quatro categorias: nomes, verbos, adjectivos e advérbios. Os *synsets* são utilizados no REBUILDER para classificarem objectos de software. A cada objecto corresponde um *synset* de contexto que representa o significado do objecto. Este *synset* de contexto pode ter várias aplicações, por exemplo calcular a distância semântica entre dois objectos.

Os casos são armazenados em ficheiros, isto faz com que o acesso aos casos seja mais lento do que se estes estivessem em memória. Para resolver este problema são utilizados índices para indexar os casos e os objectos que o caso contém. Estes possibilitam o acesso aos casos relevantes sem ter que ler todos os casos para memória. A cada objecto de um caso é atribuído um índice. Este índice não é mais do que o *synset* de contexto do objecto, que é depois associado ao respectivo *synset* do WordNet. A informação que fica associada ao *synset* é: nome do objecto, tipo de objecto, e nome do ficheiro que contém o objecto.

A taxonomia de tipos de dados é uma hierarquia de tipos de dados utilizados pelo REBUILDER. Estes são usados na definição de atributos ou parâmetros de métodos. Esta taxonomia tem como função o cálculo da distância conceptual entre dois tipos de dados.

Gestor da Base de Conhecimento

O gestor da BC é usado pelo administrador para gerir a BC, mantendo-a actualizada e coerente. Este módulo para além de ter todas as funcionalidades do editor de UML, acrescenta ainda várias funções de gestão da BC. A lista de funções acrescentadas é a seguinte:

- Comandos sobre a BC:
  - Criar uma Base de Casos.
  - Abrir uma Base de Casos.
  - Fechar uma Base de Casos.

- O comando *Gestor da Base de Casos* permite aceder ao conteúdo da base de casos, podendo o administrador:
  - Adicionar casos.
  - Remover casos.
  - Modificar casos.
  - Mudar o estado de um caso.
- Um comando que activa a aprendizagem de casos. Este permite ao administrador analisar o conteúdo da base de casos. Para isso o administrador tem ao seu dispor um conjunto de estratégias de manutenção da base de casos que o auxilia a escolher quais os casos que devem fazer parte da base de casos e quais os que não devem.
- O comando *Configurações* adiciona parâmetros de configuração aos já presentes no editor de UML, versão usada pelos *designers* de software. Permite também ao administrador configurar os mecanismos de raciocínio.

As responsabilidades básicas do administrador são a gestão da base de casos e a configuração do sistema. Na gestão da base de casos insere-se a revisão dos *designs* submetidos pelos *designers* de forma a transformá-los em casos confirmados.

Motor de Raciocínio Baseado em Casos

O motor de RBC executa as tarefas que envolvem raciocínio sobre os casos, e é composto por seis módulos (ver figura 2). Cada um implementa uma fase do ciclo de RBC. Estes módulos podem ser combinados de diferentes maneiras, sendo talvez a mais comum, apresentada na figura 4.

Seguidamente os vários módulos são descritos de uma forma sucinta (para informação mais detalhada ver capítulo 4).

O módulo de recolha selecciona da base de casos um conjunto de casos ordenados por grau de semelhança com o problema especificado pelo *designer*. Permite ao *designer* avaliar o conjunto de casos mais semelhantes com o problema, de forma a poder explorar diversos *designs* alternativos, ou pode ainda reutilizar algum destes *designs*. Este módulo funciona como um assistente de busca inteligente, que primeiro

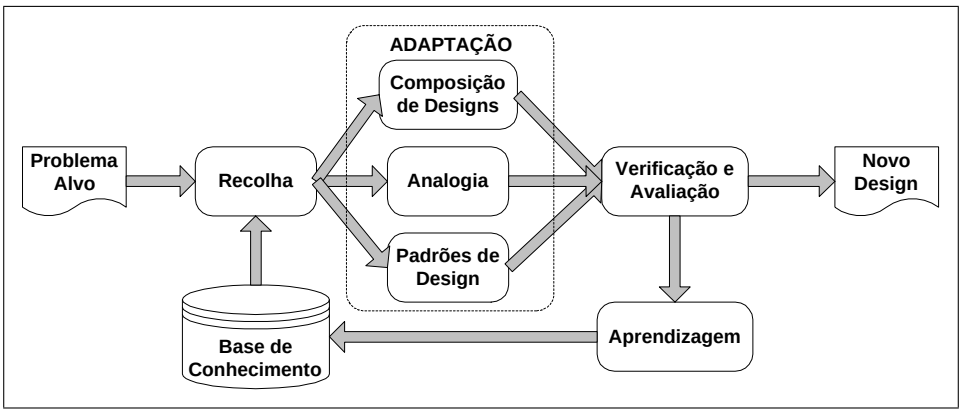


Figure 4: Uma maneira possível de combinar os diferentes módulos de raciocínio.

recolhe um conjunto de casos relevantes usando os índices do WordNet, e depois ordena-os por semelhança com o problema, usando métricas de semelhança. Estas métricas avaliam três tipos de semelhança: funcional, comportamental e estrutural. Um problema é um objecto UML que pode ser de um de três tipos: *package* (correspondendo a um diagrama), classe e interface. Para cada um destes tipos de objectos existe um algoritmo para recolha de casos, que utiliza o *synset* do objecto problema para fazer a pesquisa no WordNet, procurando índices de objectos do mesmo tipo. Depois é usada uma métrica de semelhança específica para cada tipo de objectos, de forma a ordenar a lista de objectos recolhidos. No final, a lista ordenada resultante é apresentada ao *designer*.

A analogia no REBUILDER é usada para gerar novos *designs*. Este mecanismo tem três fases: selecção, mapeamento, e transferência. Na primeira fase vão ser seleccionados os casos que vão servir de base à analogia. Esta fase, primeiro selecciona um conjunto de casos através do algoritmo de recolha, depois vai ordená-los usando para isso uma das três métricas possíveis: métrica de semelhança do mecanismo de recolha, métrica de semelhança que usa uma heurística que avalia a independência de objectos, ou métrica que se baseia exclusivamente em propriedades estruturais dos casos candidatos. A fase de mapeamento estabelece um mapeamento entre os objectos do problema alvo e os do candidato seleccionado. Existem dois algoritmos que podem ser utilizados para este efeito: um baseado em relações, e outro baseado em objectos. Tendo o mapeamento sido efectuado, a fase seguinte é a transferência de conhecimento do caso candidato para o problema alvo, usando os mapeamentos estabelecidos. O resultado é um novo *design* gerado por analogia que é apresentado

ao *designer* como solução para o problema especificado.

O módulo de composição de *designs* gera novos *designs* a partir de um ou mais casos. Existem duas estratégias de composição: baseada no melhor caso e baseada no melhor conjunto de casos complementares. A primeira selecciona o caso mais semelhante com o problema alvo da base de casos usando o módulo de recolha. Depois, o caso seleccionado vai ser mapeado com o problema alvo. Enquanto existirem objectos no problema alvo por mapear, o algoritmo vai utilizar o módulo de recolha para ir buscar casos que contenham os objectos ainda não mapeados, de forma a mapeá-los. Depois de estarem todos os objectos do problema mapeados, ou então não existirem mais casos que possam ser utilizados, o algoritmo vai transferir conhecimento dos casos utilizados para o problema alvo, usando os objectos mapeados. O resultado é um novo diagrama que foi gerado por composição dos casos recolhidos. A segunda estratégia de composição baseia-se na construção de conjuntos de casos complementares. Para isto, o algoritmo vai recolher todos os casos da base de casos que têm pelo menos um objecto em comum com o problema. Depois são formados todos os conjuntos de casos possíveis, utilizando os casos recolhidos. Para cada um destes conjuntos é calculado em que grau é que o problema é mapeado, sendo depois estes conjuntos ordenados pelos valores obtidos. Os casos do conjunto com o melhor valor são depois utilizados para transferir o conhecimento para o problema, de forma a construir um novo diagrama. O novo diagrama é depois apresentado ao *designer*.

Os padrões de *design* no REBUILDER podem ser usados para gerar novos diagramas. A ideia central deste módulo é melhorar um diagrama existente (o problema alvo) de acordo com um padrão de *design*. Este módulo vai utilizar um tipo de casos diferente, chamados de casos DPA (*Design Pattern Application*). Estes casos descrevem uma situação aonde um determinado padrão de *design* foi utilizado. O algoritmo vai seleccionar o caso DPA mais semelhante com o problema alvo. Depois vai aplicar o padrão de *design* do caso DPA seleccionado para modificar o problema alvo de acordo com o padrão de *design*. O resultado é um novo diagrama melhorado de acordo com a filosofia do padrão escolhido.

O módulo de verificação e avaliação tem como objectivos testar a coerência de diagramas, e fazer a sua avaliação. A verificação pode ser utilizada no REBUILDER de três formas distintas: ser solicitada pelo *designer*, no fim do processo de analogia sobre o diagrama gerado, ou então no fim do processo de composição de *designs* sobre

o diagrama gerado. Na verificação são utilizadas quatro fontes de conhecimento de forma a tentar identificar elementos de um diagrama que são considerados incorrectos, estas são por ordem de prioridades: casos de verificação, WordNet, casos de *design*, e *designer*. Os casos de verificação são um terceiro tipo de casos do REBUILDER que descrevem situações em que um elemento de um diagrama foi considerado errado ou correcto. Desta forma o sistema quando se depara com uma situação semelhante pode determinar qual a acção a tomar: eliminar o elemento ou considerá-lo correcto. Se não existir na base de casos nenhum caso de verificação que possa ser usado para verificar um determinado elemento do diagrama, então o sistema vai procurar no WordNet uma relação semântica que possa verificar o elemento. Se este não tiver sucesso, então a fonte de conhecimento seguinte a ser utilizada vai ser a base de casos de *design*. Se existir num caso de *design* um elemento semelhante para uma situação semelhante, então o elemento é considerado válido. Senão, o último recurso de verificação é perguntar ao *designer* qual a validade do elemento que está a ser verificado. Consoante a resposta do *designer*, assim é criado um caso de verificação de sucesso ou de falha. A avaliação é um processo mais simples, aonde o diagrama vai ser avaliado com base num conjunto de métricas de engenharia de software e no conjunto de propriedades do diagrama.

O módulo de aprendizagem é responsável pela acquisição de novos casos para a base de casos. Para o efeito dispõe de um conjunto de políticas de manutenção da base de casos que podem ser usadas pelo administrador da base de conhecimento para determinar como deve evoluir o conteúdo da base de casos ao longo da resolução de problemas. As políticas implementadas têm origem em trabalhos da literatura de RBC, sendo que tiveram que ser adaptadas à representação de casos do REBUILDER.

Existe ainda um outro módulo no motor de RBC, mas que se considera como sendo um módulo transversal, que é o módulo de desambiguação de palavras e que não está representado na figura 4. Este módulo tem como objectivo encontrar o *synset* correcto para um determinado objecto (*package*, classe ou interface). Para isso, este módulo implementa um conjunto de métodos de desambiguação, alguns baseados em trabalhos da literatura de processamento de linguagem natural que foram adaptados para a desambiguação de diagramas, e outros originais desta tese.

A secção seguinte apresenta as principais conclusões dos estudos experimentais

efectuados, estabelecendo linhas orientadoras quanto às configurações a dos algoritmos a usar em cada situação no REBUILDER.

Experimentação

Esta secção faz um breve resumo dos principais resultados experimentais obtidos com o REBUILDER. O capítulo 5 apresenta todos os detalhes sobre a experimentação. O trabalho experimental efectuado tem como objectivo avaliar e traçar um perfil de performance de cada um dos módulos do motor de RBC, bem como deixar pistas para a parametrização e escolha de entre os algoritmos de raciocínio disponíveis no REBUILDER.

Em quase todos os testes foi utilizada a mesma base de conhecimento que designamos por KB 1.1. Esta contém um conjunto de problemas, a base de casos, o WordNet, os índices dos casos e a taxonomia de tipos de dados. A versão do WordNet que foi usada é a 1.7.1, que tem cerca de oitenta mil *synsets* de nomes e cem mil relações semânticas. A base de casos tem 60 casos, cada um representando um diagrama de um sistema de software. Os casos repartem-se por quatro domínios: sistemas de informação para a banca, sistemas de informação para a área da saúde, sistemas de informação para a área da educação, e sistemas de informação para a área comercial. Cada caso é composto por uma *package* com 5 a 20 classes (o número total de classes é de 586). Cada classe tem até 20 atributos e 20 métodos. Estes diagramas estão definidos ao nível conceptual, de forma que o *design* está numa fase genérica tendo apenas as classes, atributos e métodos fundamentais.

Nas experiências utilizaram-se quatro conjuntos de problemas. Três destes conjuntos são constituídos por diagramas que resultam de casos da base de casos, em que parte dos objectos foram removidos. Estes conjuntos são designados por P20, P50 e P80. O conjunto P20, com 25 problemas, é composto por diagramas que são a cópia dum caso só que com 80% dos objectos, atributos e métodos eliminados. Os conjuntos P50 e P80 têm, respectivamente, 50% e 20% de objectos eliminados. O quarto conjunto de problemas designado por *PVar*, contém 25 problemas diferentes dos casos na base de casos, no entanto, dentro do mesmos domínios da base de casos.

Recolha de Casos

Várias experiências foram efectuadas para testar a recolha de casos. Na primeira experiência testou-se a performance de recolha de diagramas (recolha de *packages*). Foram utilizados os conjuntos de problemas P20, P50 e P80. Para cada problema destes conjuntos pré estabeleceu-se qual o caso mais semelhante e quais os casos relevantes. Esta experiência foi utilizada para explorar o mecanismo de recolha, procurando identificar erros e limitações deste módulo, bem como melhoramentos a fazer. Dos resultados obtidos pode concluir-se que a configuração que dá mais importância à semelhança funcional obteve os melhores resultados. Em relação ao tamanho do problema, a tendência observada é de que para problemas maiores a recolha tem conseguido melhores resultados de precisão, quando é dada mais importância à semelhança estrutural. Este era um resultado esperado.

Juntamente com estas experiências iniciais, foram feitas experiências para testar o mecanismo de recolha de classes, sendo a experiência semelhante à anterior (só que para classes). Os resultados experimentais apontam para uma grande importância da semelhança funcional entre classes. Sendo assim, a classificação correcta dos objectos é fundamental. Para poder avaliar o grau de erro introduzido na precisão da recolha, foi efectuado um estudo experimental. Este estudo consistiu em comparar o grau de precisão do mecanismo de recolha com diferentes graus de erros nas classificações dos objectos. Os resultados obtidos mostram que para uma base de casos sem objectos mal classificados, o valor da recolha de casos é de 85.12%, enquanto que este valor para uma base de casos com 16% de objectos mal classificados é de 58.11%. O que se traduz numa diminuição da capacidade de recolha de 31.73%.

Depois destas primeiras três experiências, foram operadas algumas mudanças e melhoramentos no mecanismo de recolha. Dentro dos melhoramentos mais importantes, está o desenvolvimento de duas novas versões do algoritmo de recolha, ambas com o objectivo de lidar com a classificação errada de objectos. A métrica de semelhança também sofreu melhoramentos. Com estas alterações foram feitas mais experiências, onde foi usado o conjunto de problemas *PVar*. Estas experiências tinham o objectivo de testar o mecanismo de recolha em termos de precisão, nível de recolha de elementos relevantes, avaliação das três versões do algoritmo, pesos para a métrica de semelhança para *packages*, e o tamanho do conjunto de casos a recolher.

Os resultados obtidos indicam que um dos novos algoritmos de recolha obtêm os melhores resultados. Este algoritmo utiliza não só o *synset* da *package* alvo para iniciar a recolha, mas também os *synsets* dos objectos do problema, aumentando assim a tolerância ao ruído por parte do algoritmo. Em termos da melhor configuração de pesos para a métrica de semelhança de *packages*, a combinação que dá mais importância à semelhança estrutural é melhor do que a que dá mais importância à semelhança funcional. O tamanho do conjunto de casos recolhidos indicado para optimizar a precisão / nível de recolha é o valor 5. Em termos de escalabilidade, os resultados mostram que o tempo de computação por caso recolhido desce entre 75% a 85%, entre a situação com um caso recolhido e a com 20 casos recolhidos.

Analogia

Para testar o mecanismo de analogia foram efectuadas três experiências. A primeira experiência tinha um cariz exploratório da performance dos dois algoritmos de mapeamento desenvolvidos e do parâmetro que define o limiar de mapeamento entre dois objectos. Dos resultados obtidos podemos concluir que o algoritmo de mapeamento baseado em objectos tem um melhor desempenho do que o algoritmo baseado em relações. Em termos do limiar de mapeamento entre dois objectos, os melhores valores foram obtidos com o limiar 0.8 para o algoritmo de mapeamento baseado em objectos, e entre 0.7 e 0.8 para a versão baseada em relações.

A segunda experiência visa avaliar a integração do mecanismo de recolha com o de analogia. O mecanismo de recolha é responsável pela selecção dos candidatos para analogia, e vai ser testado em relação à sua eficiência, de forma a ser escolhida a melhor forma de seleccionar os candidatos para a analogia. Os resultados mostram um claro compromisso entre tempo de computação e qualidade dos resultados. O uso da recolha de casos torna a selecção mais rápida, no entanto não é global à base de casos podendo falhar bons candidatos para a analogia.

Os testes anteriores foram efectuados antes dos melhoramentos efectuados no mecanismo de recolha. As últimas experiências foram efectuadas já com as três estratégias de recolha, que são avaliadas em conjunto com a analogia. São também exploradas algumas das propriedades criativas do mecanismo de analogia. Os resultados obtidos sugerem que a semelhança semântica deve ser tida em conta na recolha de casos para a analogia, se se quiser obter diagramas tendencialmente mais correctos.

Por outro lado a semelhança estrutural revelou-se mais importante para a obtenção de soluções mais originais. Uma estratégia de recolha que combina estes dois factores é um bom compromisso.

Composição de *Designs*

As experiências envolvendo o mecanismo de composição de *designs* tiveram vários objectivos: comparar o mecanismo de analogia com o de composição de *designs*, comparar as duas estratégias de composição de forma a determinar qual a melhor, e identificar qual o melhor valor para o limiar máximo de mapeamento para a composição de *designs*.

De uma forma geral, a analogia é mais rápida computacionalmente do que qualquer das duas estratégias de composição. Enquanto que a pior estratégia de composição é a baseada no melhor caso (BMC), ficando a baseada no melhor conjunto de casos complementares (BMCCC) entre a analogia e a BMC. O aumento do limiar de mapeamento aumenta o tempo de computação, o que é explicado pelo aumento do número de mapeamentos possíveis. A estratégia BMCCC apresentou o melhor compromisso entre a qualidade das soluções geradas (em termos de objectos mapeados e errados) e o tempo de computação para as gerar, isto em comparação com a estratégia BMC e a analogia. O limiar do número de casos recolhidos para composição mais indicado por via de experimentação é de 10, para ambas as estratégias de composição.

Padrões de *Design*

As experiências que visam avaliar o mecanismo de padrões de *design* usaram uma base de casos DPA com 60 casos gerados a partir da base de casos de *design*. Cinco padrões de *design* foram implementados e testados: *Abstract Factory*, *Builder*, *Composite*, *Singleton* e *Prototype* (para uma descrição detalhada destes padrões consultar [Gamma et al., 1995]). Nestas experiências foram usados os diagramas do conjunto de problemas *PVar*. As experiências consistiam em avaliar se o módulo de padrões de *design* aplicava correctamente os padrões, gerando diagramas coerentes. Os resultados obtidos mostram que o mecanismo de recolha de casos DPA consegue seleccionar 76% de casos relevantes, recolhendo 3 casos DPA. As indicações obtidas por estas

experiências são boas e permitiram também identificar algumas limitações do mecanismo. Uma observação mais atenta das situações em que o mecanismo falhou revela que falta algum conhecimento na descrição dos DPAs, tais como, a *package* a que o objecto pertence.

Verificação e Avaliação

A capacidade do mecanismo de verificação para corrigir erros do *design* foi testada com as soluções geradas pelo mecanismo de analogia e a composição de *designs*. O objectivo principal era determinar o grau de melhoramento que o mecanismo de verificação trazia à qualidade das soluções geradas por estes dois módulos, o que pode ser avaliado pelo número de objectos considerados errados ou incoerentes com o respectivo diagrama. Dos resultados obtidos, o número de objectos errados nas soluções geradas por composição de *designs* baixou em 36% e na analogia baixou 86%. Estes números vêem demonstrar que o mecanismo de verificação é bastante útil em conjunto com os mecanismos de adaptação.

Aprendizagem

A experimentação feita ao módulo de aprendizagem incidiu sobre as políticas de manutenção da base de casos. As experiências tiveram como objectivos principais: análise da evolução temporal e nível de competência alcançado pela base de casos obtida por cada uma das estratégias.

Os resultados da análise temporal mostram que existem três grupos de estratégias, em termos de nível de desempenho (a definição das estratégias estão na secção 4.6, a secção 5.7 apresenta os resultados experimentais em maior detalhe). Sendo que num dos grupos, o tempo de computação das estratégias que compreende é demasiado alto para serem usadas. Os restantes grupos mostraram ter tempos de computação baixos, chegando ao ponto de um dos grupos apresentar tempos de computação muito baixos e independentes do número de casos da base de casos.

A análise dos resultados obtidos em relação ao desempenho de cada uma das estratégias mostra que a maior parte delas faz baixar a competência da base de casos com a diminuição dos casos na base de casos. No entanto, a maior parte das estratégias usadas só começa a provocar perdas consideráveis de competência da

base de casos, quando o número de casos é reduzido para metade ou menos. Isto pode indicar que a base de casos tem muitos casos redundantes, e que as estratégias estão a funcionar correctamente escolhendo bem os casos a serem eliminados, até um determinado ponto. Dos resultados obtidos pode ainda ser inferido que o mecanismo de adaptação faz uma grande diferença em relação à capacidade de resolução de problemas do REBUILDER. A competência de uma base de casos com apenas 10 casos e com os mecanismos de adaptação disponíveis consegue atingir valores entre 68% e 92%, dependendo da estratégia de manutenção escolhida. Para uma base de casos com 60 casos mas apenas com a recolha de casos como mecanismo de resolução de problemas, os valores da competência só chegam aos 48%.

Desambiguação de Diagramas

Esta secção descreve duas experiências feitas com o módulo de desambiguação de diagramas. A primeira experiência é exploratória e pretende testar e avaliar o primeiro método de desambiguação utilizado (ver secção 4.7 para uma descrição completa de todos os métodos de desambiguação utilizados). Desta experiência adveio uma necessidade de implementar e testar outros métodos de forma a tentar melhorar os resultados. Assim a segunda experiência é feita já com vários métodos de desambiguação implementados.

O primeiro método de desambiguação implementado baseia-se na distância semântica entre os *synsets* candidatos e os *synsets* dos objectos de contexto. Nesta experiência explorou-se qual o contexto que melhor servia para desambiguar um objecto num diagrama. Chegou-se à conclusão de que os melhores resultados eram obtidos com todos os objectos do diagrama como contexto. Foram ainda testadas várias métricas de cálculo da distância semântica. Aquela com melhores resultados utiliza todas as relações semânticas para calcular a distância. Os melhores resultados obtidos estão na gama dos 71.18%, o que segundo Gale [Gale et al., 1992] está entre os valores tidos como esperados para um mecanismo de desambiguação, que são entre 68% e 96.8%.

Na segunda experiência que envolveu todos os métodos desenvolvidos, os resultados obtidos para a desambiguação de diagramas aumentou ligeiramente, chegando quatro dos métodos testados ao valor de 74.53%. Foram feitas mais experiências envolvendo outro tipo de desambiguação, mas aqui os resultados não foram promissores.

Trabalhos Relacionados

Esta secção faz um breve resumo da comparação que é feita da abordagem desenvolvida com outros sistemas cujo trabalho está de alguma forma relacionado com o REBUILDER. Os sistemas seleccionados são aqueles que se enquadram dentro do tipo de sistema do REBUILDER, nomeadamente: sistemas de RBC para *design*, sistemas de RBC para reutilização de software, sistemas de reutilização de software que usam analogia, e outros sistemas de reutilização de software. Para uma descrição mais detalhada destes sistemas consultar o capítulo 6.

Sistemas de Raciocínio Baseado em Casos para *Design*

As semelhanças do REBUILDER com outros sistemas de RBC[17] são reduzidas. No entanto, vamos destacar as principais diferenças e semelhanças destes sistemas com o REBUILDER. A maior parte destes sistemas usa raciocínio baseado em modelos (RBM), para além de RBC. Uma possível justificação são os formalismos de representação de casos usados, que permitem este tipo de raciocínio mais complexo e dependente de conhecimento do domínio. Uma das diferenças principais do REBUILDER relativamente a estes sistemas é o uso de analogia dando ao sistema capacidade para gerar novas soluções potencialmente criativas, e o processamento de linguagem natural (PLN) de forma a associar objectos de software a conceitos.

A maior parte dos sistemas estudados usa uma representação de casos baseada em pares atributo/valor ou em modelos. Uma representação baseada em pares atributo/valor tem a vantagem de ser simples de manipular, no entanto, tem limitações quanto ao poder expressivo. Os modelos por seu lado, têm uma grande capacidade expressiva, mas geralmente a complexidade dos mecanismos de raciocínio que os manipulam é grande. Em relação à representação baseada em diagramas de UML, estas

[17] Os sistemas com que o REBUILDER foi comparado são: EADOCS [Netten and Vingerhoeds, 1996], COMPOSER [Purvis and Pu, 1995, Pu and Purvis, 1997, Purvis and Pu, 1998], CADET [Sycara and Navinchandra, 1991, Sycara and Navinchandra, 1993], CADRE [Dave et al., 1991], IDIOM [Smith et al., 1995], FABEL [Voss et al., 1994, Coulon and Gebhardt, 1994, Gebhardt et al., 1997], TOPO [Voss and Coulon, 1996], CASECAD, CADSYN, WIN, DEMEX [Maher et al., 1995, Maher, 1996], KRITIK family of design systems [Goel, 1991, Goel, 1992, Bhatta and Goel, 1993a, Bhatta and Goel, 1993b, Goel et al., 1997a, Goel et al., 1997b, Bhatta and Goel, 1997a, Bhatta and Goel, 1997b], IM-RECIDE [Bento, 1996, Gomes et al., 1996, Gomes et al., 1998] e CREATOR [Gomes and Bento, 1997b, Gomes and Bento, 1998].

têm como principal vantagem uma elevada capacidade de comunicação e de familiaridade com o *designer* de software. Em termos de interpretação semântica para o *design* de software, uma representação baseada em modelos é mais complicada de interpretar do que os diagramas de classe de UML.

A estrutura de indexação é crucial para uma recolha rápida e eficaz. A maior parte dos sistemas de RBC para *design* usa estruturas hierárquicas para a organização dos índices. Esta estrutura provou ser bastante eficaz na recolha de casos, daí ser bastante usada. O REBUILDER segue este tipo de abordagem usando o WordNet como estrutura de indexação. A maior parte dos sistemas usa indexação de casos com base em características funcionais, no entanto existem alguns sistemas que usam características estruturais e comportamentais para indexar casos. No REBUILDER, a indexação é feita com base nas funcionalidades usando os *synsets*. No entanto, em termos da métrica de semelhança, o REBUILDER já usa características estruturais e comportamentais dos casos para os ordenar por grau de semelhança com o problema alvo.

A adaptação é um dos tópicos em destaque no REBUILDER, com três tipos de adaptação implementados. A maior parte dos sistemas de RBC para *design* apenas usa um método de adaptação. Os métodos mais utilizados por outros sistemas são o RBM, raciocínio baseado em regras (RBR) e o raciocínio baseado em restrições (RBRt).

A verificação é outro aspecto que foi abordado no REBUILDER, e que apenas metade dos sistemas estudados usa. Dos que usam, grande parte recorre ao RBM para fazer uma simulações do *design*.

Sistemas de Raciocínio Baseado em Casos para Reutilização de Software

Os sistemas de RBC para reutilização de software[18] têm mais em comum com o REBUILDER do que os sistemas da subsecção anterior, devido a estarem num domínio de aplicação semelhante. A maior parte destes sistemas reutiliza código, e destes

[18] Os sistemas com que o REBUILDER foi comparado são: o sistema desenvolvido por González et. al. [Fernández-Chamizo et al., 1996, González and Fernández, 1997, González-Calero et al., 1999], Déjà Vu [Smyth and Cunningham, 1992, Smyth and Keane, 1995a, Smyth, 1996], o sistema desenvolvido por Krampe e Lusti [Krampe and Lusti, 1997], CAESER [Fouqué and Matwin, 1993], otrabalho desenvolvido por Althoff e Tautz [Althoff et al., 1997, Tautz and Althoff, 1997], e CREATOR II [Gomes and Bento, 1999b, Gomes and Bento, 2001].

apenas um reutiliza código orientado para objectos, os outros lidam com código procedimental. Esta é uma das grandes diferenças relativamente a estes sistemas, pois o REBUILDER destina-se a reutilizar *designs* de software e não código. Apenas o trabalho de Krampe e Lusti reutiliza *designs*, e fá-lo de uma forma abstracta e ligeira, pois reutiliza conhecimento de software de todas as fases de desenvolvimento.

Em termos de mecanismos de raciocínio usados, todos os sistemas apresentados usam o RBC e outro paradigma de raciocínio. Este mecanismo de raciocínio secundário é usado para complementar o RBC em tarefas como recolha de casos, adaptação e verificação. As representações de casos usadas vão desde pares atributo/valor, até representações mais complexas como modelos ou hierarquias de componentes. Estes formalismos complexos mostram-se adequados a domínios mais específicos e restritos, como é o caso do REBUILDER. Uma representação intermédia também usada é a baseada em enquadramentos[19], que pode ser vista como pares atributo/valor organizados numa estrutura. O REBUILDER utiliza uma representação baseada em grafos (diagramas de classe de UML), cuja principal vantagem é que a representação interna e externa dos *designs* é a mesma.

A estrutura de indexação usada nestes sistemas é semelhante, com a maioria dos sistemas a usarem uma indexação baseada em funcionalidades, com uma taxonomia de funções como estrutura de indexação. Isto é bastante natural, pois o problema alvo é especificado em termos de funcionalidades, e as hierarquias são uma estrutura eficiente para organizar os índices. Como consequência a recolha de casos é baseada em funcionalidades. Alguns sistemas (Déjà Vu) complementam este tipo de recolha guiando-a com base no esforço de adaptação, ou recorrendo a especificações comportamentais. No caso do REBUILDER, a recolha é feita com base em funcionalidades, tendo o WordNet como estrutura hierárquica de indexação. Já o ordenamento dos casos é feito com base em semelhanças funcionais, comportamentais e estruturais. Outras abordagens são: o trabalho desenvolvido por Althoff e Tautz, que segue uma abordagem que usa os K vizinhos mais próximos[20] para a recolha e ordenamento, e outro sistema desenvolvido por González et. al., que usa lógica descritiva para fazer a recolha.

Muitos dos sistemas apresentados utilizam regras ou heurísticas para adaptação.

[19] *Frames* em Inglês.

[20] Em Inglês *K-Nearest Neighbor* ou KNN. Método que consiste na recolha e selecção dos K casos mais semelhantes ao problema com base numa métrica de semelhança.

O REBUILDER explora mecanismos diferentes, como a analogia, a composição de *designs*, e os padrões de *design*. Enquanto que as regras e heurísticas nos sistemas referidos acima são dependentes do domínio, os mecanismos do REBUILDER são independentes do domínio, com excepção dos padrões de *design*.

Em relação à verificação, existem apenas dois sistemas (o trabalho de Krampe e o CAESER) que fazem verificação das novas soluções, para além do REBUILDER. A abordagem seguida por Krampe usa regras de integridade e modelos, enquanto que o CAESER usa grafos de causa-efeito, que podem ser interpretados como modelos. Neste aspecto o REBUILDER usa três mecanismos diferentes: casos de verificação, casos de *design* e o WordNet.

Sistemas de Reutilização de Software que usam Analogia

Um característica comum à maior parte dos sistemas de reutilização de software que usam analogia[21] como mecanismo de raciocínio, é que reutilizam código. Apenas um sistema lida com a fase de análise (o trabalho de Maiden) e outro com a fase de *design* (o sistema ROSA de Tessem), tal como o REBUILDER.

A representação de conhecimento é variada e depende de sistema para sistema, sendo que a maior parte dos sistemas utiliza uma linguagem de representação específica para descrever o código ou o *design*. Neste aspecto o REBUILDER usa uma linguagem de âmbito geral e que se está a tornar uma linguagem padrão na especificação de sistemas de software.

A selecção de candidatos para analogia é feita em geral com base na semelhança semântica. Em alguns dos sistemas (os trabalhos de Spanoudakis, Jeng e Harandi) considerados utilizam também a semelhança estrutural. Esta semelhança é também usada no algoritmo de recolha ou na métrica de semelhança usada para ordenar os candidatos. Para além da semelhança semântica (neste caso vertente funcional) e estrutural, o REBUILDER utiliza também a semelhança de comportamento no ordenamento dos casos.

[21] Os trabalhos comparados são: Maiden e Sutcliffe [Maiden and Sutcliffe, 1992], Spanoudakis [Spanoudakis and Constantopoulos, 1994], Jeng e Cheng [Jeng and Cheng, 1993], Harandi e Bhansali [Harandi and Bhansali, 1998], e Tessem (o sistema ROSA) [Tessem et al., 1994].

Outros Sistemas de Reutilização de Software

A grande maioria dos sistemas de reutilização de software estudados no âmbito deste trabalho, exceptuando aqueles que estão nas categorias anteriores, focam a reutilização de código[22]. Alguns destes sistemas também fazem reutilização de *designs*, no entanto, esta é um subproduto da reutilização de código, pois as várias classes que são reutilizadas em bloco são vistas como parte do *design*. A maioria destes sistemas reutiliza código orientado para objectos, exceptuando o sistema RSL e o trabalho de Prieto-Díaz que lidam com qualquer tipo de código (ver secção 6.4). Na maior parte destes sistemas apenas é feita recolha de código, o que permite que usem um formalismo de representação simples de manipular, como pares atributo/valor ou enquadramentos. Representações mais complexas para a representação de código, como grafos, não são exploradas no seu potencial por alguns destes sistemas. Por exemplo, no caso do REBUILDER, a representação usada permite um conjunto alargado de mecanismos de raciocínio, como é demonstrado neste trabalho.

Em relação à recolha, existem quatro formas de recolha identificados nos sistemas analisados nesta secção: baseada em atributos, semântica, com base em linguagem natural, e estrutural. A recolha baseada em atributos utiliza os atributos que descrevem o código de forma a efectuar a procura no repositório. Este método não distingue se os atributos são de cariz funcional, comportamental ou estrutural, o que pode ser uma vantagem devido à natureza não restrita da recolha. No entanto, pode ser também uma limitação, pois para os mecanismos de raciocínio os atributos não são tipificados, o que seria útil para estabelecer certo tipo de inferências. O segundo tipo de recolha, recolha semântica, usa uma classificação funcional dos componentes como forma de seleccionar os componentes relevantes para um determinado problema. A recolha baseada na linguagem natural, na maior parte das vezes, utiliza palavras chave ou conceitos para indexar os componentes. Finalmente, a recolha estrutural é baseada na estrutura dos componentes, e regra geral, é mais complexa do que os outros tipos de recolha. O REBUILDER usa a recolha funcional ou semântica para seleccionar um primeiro conjunto de casos relevantes, e depois utiliza uma métrica

[22]Os trabalhos e sistemas comparados nesta subsecção são: o trabalho de Prieto-Díaz [Prieto-Diaz and Freeman, 1987, Prieto-Diaz, 1991], o trabalho de Liao [Liao et al., 1999], o sistema RSL [Burton et al., 1987], o trabalho de Borgo [Borgo et al., 1997], o trabalho de Helm [Helm and Maarek, 1991], o sistema ROSA [Girardi and Ibrahim, 1994] de Girardi, o trabalho de Burg e Riet [Burg and Riet, 1997], o sistema LaSSIE [Devanbu et al., 1991], e o trabalho de Bassett [Bassett, 1987, Bassett, 1997]

baseada na semelhança funcional, comportamental e estrutural, para ordenar os casos seleccionados.

Conclusões

Esta secção apresenta as contribuições desta tese, que são o resultado de um trabalho de três anos. As principais áreas de contribuição inserem-se nos seguintes temas: Raciocínio Baseado em Casos, Reutilização de Software, ferramentas ICASE, Desambiguação de Palavras, Analogia e Sistemas de *Design* Criativo. Durante o desenvolvimento do REBUILDER foram surgindo várias questões científicas relevantes para este trabalho. No entanto, a falta de recursos, nomeadamente tempo, e os objectivos delimitados desta tese, não permitiram a sua exploração, sendo deixados como trabalho futuro ou direcções para investigação futura. Na secção seguinte apresentamos vários desafios de investigação e melhoramentos a efectuar no REBUILDER.

Contribuições da Tese

O REBUILDER constituiu um sistema de teste para várias ideias, e um desafio que abrangeu várias áreas de investigação. O nosso objectivo inicial era o de desenvolver um sistema capaz de reutilizar *designs* de projectos anteriores. Esta ideia amadureceu e evoluiu para um sistema com uma arquitectura cliente-servidor, baseado em RBC como principal forma de raciocínio e metodologia de gestão de conhecimento. O REBUILDER pode ainda ser visto como uma ferramenta de *design* e reutilização de software que possibilita a reutilização a um nível empresarial, funcionando como repositório de *designs*. Aspectos de comunicação entre o *designer* e o sistema tiveram prioridade no desenvolvimento do REBUILDER. Estes aspectos justificam o uso da linguagem UML e a incorporação de processamento de linguagem natural.

Apresentamos agora o que entendemos serem as principais contribuições deste trabalho:

- A arquitectura cliente/servidor desenvolvida, bem como a maneira como se dividem e interagem os vários módulos de forma a integrarem a vertente ICASE com a vertente de gestão de conhecimento de *design* de software.

- A integração do WordNet na base de conhecimento e consequentemente nos

processos de raciocínio. O WordNet é usado no REBUILDER como recurso léxico, mas também como ontologia.

- A introdução de um administrador da base de conhecimento que vem permitir uma gestão coerente do conteúdo da base de conhecimento.
- A representação de casos centrada no utilizador, ou seja, o sistema utiliza a 'linguagem' do *designer* de software (o UML) em vez de ser o oposto.
- Um esquema de indexação baseado no WordNet em que não só os casos são indexados mas também os objectos dos casos, o que permite uma maior flexibilidade de recolha de casos. Outra contribuição são os algoritmos de recolha e as métricas de semelhança desenvolvidas.
- A exploração de diversos mecanismos de adaptação, resultando no desenvolvimento de três formas diferentes de adaptação de diagramas de casos antigos a um novo problema:
  - Adaptação de diagramas de classe através de raciocínio analógico. Existem várias contribuições neste ponto, nomeadamente a forma como o WordNet é utilizado para selecionar os objectos candidatos a mapeamento, a forma como os mapeamentos são efectudados e o trabalho de experimentação desenvolvido sobre o estudo do *design* criativo.
  - Outro tipo de adaptação que envolve a utilização de vários casos para solucionar um problema, que designámos por composição de *designs*.
  - A utilização de padrões de *design* de software para a modificação e evolução de *designs*. Esta contribuição é uma das mais originais pois é a primeira abordagem que automatiza todo o processo de aplicação de padrões de *design*, desde a escolha do padrão a aplicar até à aplicação do padrão escolhido.
- A utilização de casos de verificação que representam conhecimento sobre o domínio. Estes vão servir para identificarem erros em diagramas e também para os corrigirem. Para além dos casos de verficação, outra contribuição deste trabalho é a forma como cada *designer* tem a sua base de casos de verificação específica, o que permite a personalização deste mecanismo. Outro aspecto inovador é a utilização do WordNet para fazer verificação.

- No domínio da manutenção da base de casos este trabalho adapta várias estratégias de manutenção de casos para trabalharem com uma representação de casos baseada em grafos.

- Em relação ao problema de desambiguação de palavras, para além de desenvolvermos dois métodos de desambiguação, foi feito um estudo alargado comparando vários métodos descritos na literatura.

Trabalho Futuro

Durante estes três anos de trabalho foram encontradas várias oportunidades para trabalho de investigação. Algumas destas vias não foram seguidas, ou porque se encontravam fora do âmbito deste trabalho, ou porque exigiam recursos humanos mais alargados. No entanto, fica aqui um conjunto de questões que abarcam futuras direcções de investigação ou melhoramentos a efectuar posteriormente no REBUILDER.

- A integração de outras fases do processo de desenvolvimento de software no REBUILDER, que não o *design*, é um dos objectivos futuros. Nomeadamente a reutilização de diagramas de casos de uso na fase de análise, e a reutilização de código na fase de implementação. A integração da reutilização de código vem permitir que as mudanças feitas ao nível do *design* sejam automaticamente propagadas para o código e vice-versa. A fase de *design* pode ainda ser melhorada através do raciocínio com base em outros tipos de diagramas.

- Outro melhoramento futuro para o REBUILDER, será a integração de um módulo de processamento de linguagem natural, cuja função será a conversão de um texto escrito em Inglês pelo *designer*, num primeiro diagrama de classes que modeliza os principais conceitos do texto (parte deste módulo já se encontra desenvolvida).

- Como ferramenta CASE, o REBUILDER tem um comportamento reactivo. Uma ideia possível de explorar no futuro seria a implementação de um comportamento mais proactivo por parte do REBUILDER. Através de uma modelização do estado cognitivo do *designer*, seria possível para o sistema fazer sugestões de acordo com as expectativas do utilizador.

- Um outro melhoramento seria a integração de conhecimento especializado no WordNet, por exemplo, através do administrador da base de conhecimento que poderia inserir, apagar ou modificar conteúdos no WordNet. Neste ponto é de lembrar que no âmbito deste trabalho foi adicionado ao WordNet a hierarquia de classes do Java de forma a tornar o WordNet mais completo para o domínio da *design* orientado para objectos.

- Novos mecanismos de reutilização de casos podem ser facilmente integrados no REBUILDER. Adaptação baseada em modelos, regras, restrições, algoritmos genéticos, ou outro tipo de mecanismo podem ser facilmente integrados no motor de RBC.

- Um melhoramento de que o mecanismo de analogia do REBUILDER pode tirar partido, é a inclusão de critérios de avaliação das soluções geradas. Estes critérios iriam avaliar o grau de certas propriedades criativas dos *designs* gerados, de forma a melhorar a qualidade destas, tendo como objectivo a exploração de novas áreas do espaço de *design*.

- A verificação de classes e interfaces é abordada de forma parcial neste trabalho. A única verificação efectuada é feita com base no nome do objecto. No entanto, poderia utilizar-se os atributos e métodos do objecto, ou ir mais além e ter como base os outros objectos do mesmo diagrama de forma a conseguir inferir se o objecto deve estar no diagrama. Para além de que o mecanismo de recolha e selecção destes casos deve ser melhorado.

- A aprendizagem de casos de verificação pode ser melhorada, através da aprendizagem de regras de verificação ou outro tipo de conhecimento genérico que permitisse generalizar princípios gerais de verificação.

- É necessário fazer mais trabalho experimental sobre as estratégias de manutenção da base de casos. O problema é conseguir uma base de casos suficientemente grande para conseguir avaliar as várias estratégias. Juntamente com estas experiências, o teste de vários critérios de subsunção[23] de casos seria bastante interessante. Outro caminho a explorar pode ser a combinação de várias estratégias de forma a conseguir descobrir sinergias entre elas.

[23] *Subsumption* em Inglês.

- A desambiguação de palavras apesar de já dar bons resultados, é ainda insuficiente. Este módulo pode ser melhorado procurando obter maior precisão na classificação de objectos. Uma direcção interessante para trabalho futuro seria a desambiguação de nomes de relações de UML. Isto permitiria ao REBUILDER ter mais conhecimento para fazer desambiguação dos nomes dos objectos ou outro tipo de inferências. A integração de outros recursos como *corpus* de texto poderia ser interessante para o melhoramento da desambiguação.

- Ainda um trabalho futuro importante para o REBUILDER seria testar a ferramenta numa empresa. Estes testes poderiam ser efectuados utilizando a metodologia GQM (*Goal-Query-Metric*) desenvolvida por [Nick et al., 1999].

Chapter 1

Introduction

This chapter presents the motivations for this work, introducing most of the terminology and concepts used along the remaining chapters. We then outline the problems and goals addressed, defining the scope of the developed work. Finally we describe the thesis organization.

The main ideas of this thesis have been published in several papers, from which we based important parts of this document. Nevertheless, this thesis describes in greater detail and with examples the issues addressed in these papers, along with other issues not covered by these documents. The publications which were used are:

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**An Approach to Software Design Reuse Using Case-Based Reasoning and WordNet**". Accepted for Publication as a Chapter in the Book ***Soft Computing in Software Engineering***, Springer.

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**REBUILDER: A Case-Based Reasoning Design Reuse Tool**". Accepted for Publication in journal of ***Engineering Intelligent Systems***.

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**Selection and Reuse of Software Design Patterns Using CBR and WordNet**". In Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (***SEKE'03***).

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**Case-Based Reuse of UML Diagrams**". In Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (***SEKE'03***).

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**Management and Reuse of Software Design Knowledge Using**

a CBR Approach". In Proceedings of the International Joint Conference on Artificial Intelligence ***IJCAI'03 Workshop: "Knowledge Management and Organizational Memories"***.

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**Human-Machine Interaction in a CASE environment**". In Proceedings of the International Joint Conference on Artificial Intelligence ***IJCAI'03 Workshop: "Mixed-Initiative Intelligent Systems"***.

- Paulo Gomes, Nuno Seco, Francisco C. Pereira, Paulo Paiva, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**The Importance of Retrieval in Creative Design Analogies**". In Proceedings of the International Joint Conference on Artificial Intelligence "***IJCAI'03 Workshop: 3rd Workshop on Creative Systems***".

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**Noun Sense Disambiguation with WordNet for Software Design Retrieval**". In Proceedings of the Sixteenth Canadian Conference on Artificial Intelligence (***AI'03***).

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2003. "**Evaluation of Case-Based Maintenance Strategies in Software Design**". In Proceedings of the Fifth International Conference on Case-Based Reasoning (***ICCBR'03***).

- Paulo Gomes, Francisco C. Pereira, Paulo Carreiro, Paulo Paiva, Nuno Seco, José L. Ferreira, Carlos Bento, 2003. "**Solution Verification in Software Design: A CBR Approach**". In Proceedings of the Fifth International Conference on Case-Based Reasoning (***ICCBR'03***).

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2002. "**Using CBR for Automation of Software Design Patterns**". In Proceedings of the European Conference on Case-Based Reasoning (***ECCBR'02***).

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2002. "**Experiments on Case-Based Retrieval of Software Designs**". In Proceedings of the European Conference on Case-Based Reasoning (***ECCBR'02***).

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2002. "**Case Retrieval of Software Designs using WordNet**". In Proceedings of the European Conference on Artificial Intelligence (***ECAI'02***).

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2002. "**Using WordNet for Case-Based Retrieval of UML Models**". In Proceedings of the STarting Artificial Intelligence Researchers Symposium (***STAIRS'02***).

- Paulo Gomes, Francisco C. Pereira, Nuno Seco, José L. Ferreira, Carlos Bento, 2002. "**Supporting Creativity in Software Design**". ***AISB'02*** Symposium "AI and Creativity in Arts and Science".

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2002. "**Experiments on Software Design Novelty Using Analogy**". European Conference on Artificial Intelligence "***ECAI'02 Workshop: 2nd Workshop on Creative Systems***".

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2002. "**Case-Based Reasoning for Reuse of Software Designs**". 6th International Conference on Knowledge-Based Intelligent Information & Engineering Systems (***KES'02***).

- Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, Carlos Bento, 2002. "**Combining Case-Based Reasoning and Analogical Reasoning in Software Design**". In Proceedings of the 13th Irish Conference on Artificial Intelligence & Cognitive Science (***AICS'02***).

- Paulo Gomes, Francisco C. Pereira, José L. Ferreira, Carlos Bento, 2001. "**Using Analogical Reasoning to Promote Creativity in Software Reuse**". ***ICCBR'01 Workshop on Creative Systems***. Vancouver, Canada.

1.1 Motivations

One problem that software development companies face today is the increasing dimension of software systems. Not only the size and complexity have boosted, but also the number of functionalities and technologies that are involved. One decade ago, the software engineering area proposed a solution for this problem based on software reuse [Prieto-Diaz, 1993, Coulange, 1997]. The idea of this approach is to reuse previously developed software in the building of new systems. Software reuse decreases the time needed to build a software system and improves its quality.

Most of the initial research works on software reuse were at the implementation level, reusing only code [Burton et al., 1987, Prieto-Diaz and Freeman, 1987]. This is one important reuse level, but there are other levels of reuse that can be very useful for software engineers, like the design, analysis, or test levels. In fact, reuse can be present at any phase of software development.

The know-how acquired by the development teams is a valuable asset that software companies possess. Each engineer or programmer learns her/his own specific knowledge, which is then reused in other projects and tasks in which s/he participates. Experienced programmers are very important to companies, in the sense that someone with experience has higher productivity. When an experienced engineer or other qualified employee leaves a company, s/he takes with her/him a corpus of knowledge, which from the company point of view, is an important knowledge loss. In a high competitive market environment, where companies strive to be alive, this problem is crucial. Another relevant aspect is the importance of sharing know-how among the company's employees increasing productivity and minimizing the losses when they abandon. Inexperienced engineers would profit a lot from the stored knowledge, project development would need less resources and product quality would improve with the availability of a corporate memory and suitable retrieval mechanisms.

Among the different software development phases proposed in the V-model (analysis, design, implementation, testing and integration) [Boehm, 1988], the design phase is only superficially explored by software reuse. This is in part explained by the different languages used in software design, and by the fact that design is a more abstract task than coding, which makes more difficult the development of design reuse approaches. Another difficulty is the lack of a strong theory in design, which makes the design task an experience-driven process, relaying a lot on the designer's know-how. But the design phase is an important phase in any software project, being the implementation phase highly dependant on the developed design.

Software design acquires additional importance as the complexity level of software increases. This also drives development teams to be more efficient and more creative in their solutions. Software designers must find new design methodologies, trying to optimize development time, processing time, required memory, and other resources. As stated in the previous paragraph, this kind of design relies more on the designer's experience and less on defined processes and methodologies. Existing methodologies are general guidelines, which can be interpreted in several ways according to the context. Like architects, software designers frequently use their experience from the development of previous systems to design new ones. Most of the mature engineering fields make the reuse of components a development rule, but in software engineering the reuse of components and/or design ideas is not easy, given the conceptual

complexity at hand. Thus, intelligent tools that support the software design task are welcome. These tools must implement common software reuse techniques, but they also have to go further, providing support for more complex reasoning abilities and exploration of new design spaces, possibly boosting more creative designs.

Creative design is a cognitive process that generates a product said to be creative when satisfying certain properties [Dasgupta, 1994]. At least four components of creativity were identified [Brown, 1989]: creative process, creative product, creative person or entity, and creative situation. Within this model, a crucial component is the creative product. Thus an important issue in developing computational models of creativity is the evaluation of the creative product. Another important issue in creativity is the process that originated the creative product. Research developed in this domain has come up with several methods used for generation of artifacts considered creative [Partridge and Rowe, 1994, Gero, 1994a, Gero and Maher, 1993, Pereira and Cardoso, 2001, Gomes et al., 1996]. Some methods associated with creative reasoning are: cross-domain transfer of ideas (analogy), combination of ideas, and exploration and transformation of conceptual spaces. Most of these methods are used in creative design [Gero, 1994b].

Most of the tools and systems developed for software reuse (see chapter 6 for more information about works on software reuse) only provide help in retrieving software entities (like classes, functions or specifications) from repositories. But reusing software involves also adaptation of the knowledge retrieved to the system being developed, which is usually left to the designer, since this is a more complex and demanding task. Analogical reasoning [Gentner, 1983, Hall, 1989, Holyoak and Thagard, 1989] appeared as a technique that can be used to overcome some problems of the adaptation phase of software reuse, since it involves transfer of ideas and solutions from other domains to the target domain. This does not only provide a way to find solutions, but gives also the opportunity to built new solutions and, sometimes, creative ones. Not only because they are novel and unexpected, but also because they can be better, simpler and more elegant. We believe analogical reasoning can achieve this on its own or with the collaboration of the software designer, providing the designer with ideas and alternatives that help to explore the solution space in a more efficient way.

CASE (Computer Aided Software Engineering) tools have been developed to help

the software engineer during software development. Most of the modern CASE tools provide assistance through all the phases of software development. These tools have the main task of automatically checking the diagram syntax and store these diagrams in repositories. A CASE tool is generally an integration of several modules: diagram editors, text editors, information repository, and other specific modules. The majority of these modules, like the editors, are functionally simple and offer great help to the designer. But there is one module that is not explored to its full potential: the information repository. The information repository is able to store several types of knowledge (textual, graphical, multimedia or other kinds), which can be very useful to designers, but it is difficult to explore if it grows up to a large dimension. To solve the problem of reusing the knowledge stored in the information repository of CASE systems, which is software development knowledge, the designer must have at her/his disposal tools capable of reusing this knowledge. We think that Artificial Intelligence techniques are able to provide such help, taking CASE tools to a new development stage: Intelligent CASE (ICASE) tools. These tools store and reuse software development knowledge in an intelligent way, providing several cognitive mechanisms that the designer can use, like: suggesting new solutions, exploring the information repository in a semantic way, checking the semantic coherence of a system's design, learning from the interaction with the designer, and many other ways.

The Unified Modelling Language (UML, [Rumbaugh et al., 1998]) is a software design specification language that uses many types of diagrams for description of different aspects of a software system. UML is one of the leading software modelling languages in the world (if not the leading one). Due to this success and because it is an excellent way for communication among software designers, we have selected it to be used in our work. From all the diagrams used in UML, we have selected to focus on class diagrams, since they are one of the most used types of UML diagrams representing the static model of a software system.

Designers usually base their reasoning on experience, which is a common form of human reasoning. In the area of Artificial Intelligence there is a subarea that uses experiences for reasoning purposes. Experiences are represented in the form of cases, which are basic knowledge blocks. This subarea is called Case-Based Reasoning (CBR, [Kolodner, 1993]) and has been growing up in the last fifteen years. CBR can be viewed as a methodology for developing knowledge-based systems [Althoff, 2001]

that uses experience for reasoning about problems. CBR main idea is to reuse past experiences to solve new situations or problems. A *case* is a central concept in CBR, and it represents a chunk of experience in a format that can be reused by a CBR system. Because CBR and design are experience-driven, we think that CBR is a suitable reasoning paradigm to be used for a computational model of the software design task.

This work started with the idea of building a system that would provide assistance to the software designer, in tasks such as exploration of the design space. Another main idea that was developed during this work, is the management of software design knowledge in the form of a corporate memory. Both ideas are complementary and are achieved in REBUILDER using CBR as the integration methodology, enabling the combination of an UML CASE tool with a Knowledge Management tool.

1.2 Goals and Contributions

Having in mind, not only the importance of design in software development, but also the difficulty of generating systems that can reuse designs, we have set our research goal in the development of a system capable of reusing software designs and manage the knowledge associated with software design.

A software system capable of reusing designs should have two main characteristics:

1. Must be able to store the software design knowledge from the entire company in a central repository, so that any software designer in the company can reuse it.

2. Must provide a tool for handling the design knowledge, and at the same time aid the designers in a more intelligent way.

This thesis presents an approach to such a system and it's implementation in a system called REBUILDER.

REBUILDER has a client-server architecture, which fits well in a distributed environment like the ones we are used to see in software development companies. This addresses the first characteristic mentioned above, while the second one is accomplished by a client module, which is an UML CASE tool. This tool works like a normal UML editor with extra functionalities that provide cognitive aid to the software designer.

The centralization of design knowledge and cognitive aid are supported in our system by a CBR approach. This is an Artificial Intelligent methodology that reuses experience to solve new situations. In the case of REBUILDER, the design knowledge is centralized in a knowledge base in the form of past experiences. These can be reused later by the client's CASE tool. While the central knowledge base allows sharing of design knowledge among the company's designers, the CASE tool enables a designer to access and reuse this knowledge. CBR fits well into this framework, because it enables the learning of new experiences. Thus the system is capable of adapting to the company type of designs or domains.

During the development of this work we addressed several research issues. Some of these contributions that we consider central in our work are:

- Integration of an ICASE tool with a software design knowledge manager. The client/server architecture that was developed and that is supported in the CBR framework, as well as the way the different modules interact, are central contributions of our work.

- Case representation is one major issue in a CBR system. Instead of choosing a computer-centered representation formalism we have used a user-centered approach, which lead us to choose UML as the representation formalism. We show that it is possible to use UML for reasoning purposes and we have presented a way of integrating it into a CBR system.

- Case indexing and retrieval are central for the performance of a CBR system. One of the goals of this work is to develop an indexing and retrieval system flexible and efficient enough to be used by a software designer through an ICASE tool. Regarding these issues we make two contributions: the indexing scheme based on WordNet that indexes not only the cases but also it's subparts, and the associated retrieval and similarity metrics.

- After retrieval, case adaptation (or case reuse) is the next step and one of the less explored in the CBR area. It is a difficult task and it requires knowledge-intensive processes. In order to provide the designer with alternative designs and to help the exploration of the design space, we have addressed this issue in a broad perspective, trying to explore several alternative approaches:

  - The use of analogical reasoning for the adaptation of diagrams to the target problem. In this area our main contributions are the way WordNet is used for choosing analogue source candidates, the mapping algorithms and the experimental work about creative design.
  - The use of design composition as another adaptation mechanism, which can generate new solutions from several cases. Our main contribution here is the composition strategies that were developed.
  - The last adaptation strategy is the automatic application of software design patterns. We think that this is one of the strongest contributions of this work, because it is the first approach that fully automates the application of software design patterns, going from the pattern selection to its application.

- The verification phase allows the detection and correction of design errors resultant from adaptation. Being one of the goals of this work to model all the phases in the CBR cycle, the integration of a revision mechanism is essential. In this area we have contributed in three ways: the definition and use of verification cases, the personalization of the verification knowledge in the form of cases, and the use of WordNet for verification purposes.

- A CBR system must take advantage of the retention phase in order to learn new knowledge (in the form of cases). It was one of our goals to integrate this phase in our approach. We have integrated and adapted several case base maintenance policies, which provide guidance for the KB Administrator in the selection of the case base contents. Our contribution is the adaptation of these policies to a graph-based case representation.

- One of the main directives in the development of REBUILDER is the importance of easing the communication between designer and system. The integration of natural language processing in the form of object name disambiguation is one of the forms of accomplishing this. Concerning this problem (also called word sense disambiguation, shortly WSD), we have contributed with two new disambiguation methods and experimental work comparing several methods from the WSD literature.

1.3 Structure

This thesis comprises an extended summary in Portuguese and seven chapters. Chapter 1 presents the motivations, goals, contributions and structure of this thesis. Chapter 2 presents the basic concepts used in this thesis. It starts by describing CBR in design, creativity in design and finishes with an overview on software design and reuse. The goal of chapter 3 is to give an overview of REBUILDER by describing it's architecture, the UML Editor module, the Knowledge Base Manager, and Knowledge Base. Chapter 4 describes the CBR engine and its submodules. It presents the main contribution of this work and the more complex part of REBUILDER. It comprises the description of seven submodules: retrieval, analogy, design composition, design patterns, verification, learning and word sense disambiguation. Chapter 5 presents several experiments that were performed to assess the performance of the various reasoning modules. Chapter 6 describes the evaluation of the work presented in this thesis, by comparison of REBUILDER with other related systems. Chapter 7 concludes this thesis by presenting the main contributions of this work. This chapter also delineates new research directions in areas like software design reuse using CBR or intelligent CASE tools.

Chapter 2

Background Knowledge

The goal of this chapter is to introduce the reader to the main subjects present in this thesis. The first section starts by describing Case-Based Design (CBD), which is the application of Case-Based Reasoning (CBR) to design tasks. It presents the design process along two perspectives - knowledge spaces and design tasks - and presents the CBR methodology. We explore several important issues in Case-Based Design, such as: case representation, case indexing, retrieval, adaptation, verification and case learning. This description provides an introduction to some important concepts in CBD and CBR. Then the focus shifts to the concepts of creativity and design. We explore creativity at a cognitive level and from the perspective defined by Margaret Boden [Boden, 1990], which enables the discussion of computational creativity. We then stress the singularities concerning creative design and analyze them from two perspectives: the creative product and the creative process, ending with some considerations about computational approaches to creative design (at an abstract level). This section also describes analogical reasoning and how it integrates into the creative process. The sections about Creativity and Creative Design are presented at a rather abstract and theoretical level, but they provide a cognitive background to the representation formalisms and reasoning mechanisms used in this work. The next section goes into a more specialized subject - Software Design and Reuse - which is the main application domain of the system developed in this thesis. We define software design and software design reuse, two important concepts for this thesis assumptions. In this section we also address two related areas in our work: software design patterns and UML.

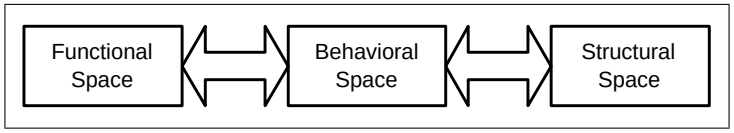


Figure 2.1: Design as a mapping process between functional, behavioral and structural spaces.

2.1 Case-Based Design

All the work done in this thesis reports to design in some way, so, it is important to analyze it from the cognitive and computational perspectives. This section starts by defining the design activity in terms of design knowledge spaces and involved tasks, and then describes how CBR can be applied to design, starting by explaining what is CBR. Then it explores more specific issues concerning CBD.

2.1.1 Design

Design is the process of generating a structural description that complies with a set of functional specifications and constraints [Tong and Sriram, 1992]. Constraints can be structural, behavioral or resource limitations. The design product is a set of components and relations between them, making the design structure. The design process involves three main types of knowledge about the domain: functional, behavioral and structural. The mapping between these three types of knowledge is central to the design process (see figure 2.1, [Reich, 1991]). Starting from the problem space, which is most of the times the functional space, the designer maps points in this space onto points or areas in the solution space, which consists on the set of possible designs [Gero, 1994a].

Gero [Gero and Maher, 1993, Gero, 1994b] classifies Design as routine and non-routine. Routine design is defined as a class of design where all the needed knowledge for the mapping process is available to the designer. Thus, routine generated designs are instances of known classes of designs. In non-routine design, some knowledge for the mapping process is missing. When this lack of knowledge is located in the structural space, it originates a subclass of non-routine design known as innovative design. If there is knowledge missing in all the spaces it is called creative design.

Innovative design creates new designs using values for the design variables, not commonly used in previous designs. The designs generated in creative design define new classes of artifacts, thus expanding the space of known designs. Creative design will be described in more detail in section 2.2.2.

A design knowledge space comprises all the knowledge available to a specific design domain or problem [Gero and Maher, 1993]. For instance, the set of all software routines for sorting a list, establishes the design knowledge space for the software problem of sorting a list of elements. In computational terms, a design space can be the set of cases of a CBR system, the production rules of a rule-based system, or any other type of knowledge used for computational design reasoning.

When dealing with a computational approach, two types of processes need to be considered in the generation of new knowledge: search and exploration. Search is a computational process that tries to find in the solution space a design that complies with the problem constraints. By definition, search only finds solutions that are already defined in the design space. Exploration works differently from search, it creates new regions in the solution space through the modification or expansion of the solution space. Although search is generally associated with routine design and exploration is associated with non-routine design, non-routine design can also involve search.

The exploration mechanism can modify the design space by adding new design knowledge or by replacing existent design knowledge. If the design space is defined in terms of design variables (in which a design space of dimension n is defined by n design variables), then two types of changes in the design space can occur: changes in the range of values that a design variable can take, and changes in the design variables. The first transformation can occur by adding or removing items from the range of values of a design variable, thus evolving the design space with time. The second transformation is more complex, and involves changing the space dimensionality. Design variables can be added to or removed from the design space, which constitutes another type of space evolution. Both transformations can happen at the same time producing a change in the design space. The first type of space transformation is related to innovative design, while the second type of transformation is generally associated with creative design [Gero and Maher, 1993, Gero, 1994b].

As presented before, design generally deals with three types of knowledge: functional, behavioral and structural. To each type of knowledge there is an associated design space. The three types of knowledge are usually connected in the way described in figure 2.1. From one space we can usually reach the other spaces. Usually the functional space is related with the problem specifications and requirements, relating this space with the design problem space. This space is transformed along the design process most of the times. Problem specifications can be changed due to client demands, and problem reformulation due to design incoherences. These are two common ways to perform these transformations. The structural space is the design solution space, since the desired solution is a design structure that achieves the desired functionalities. The behavioral space is an intermediate space between the functional and structural spaces, making a connection between problem requirements and structural components. Most of the times the behavioral space provides an explanation of how the structural components achieve the required functionalities.

Two important observations about the analysis of the tasks involved in the design process are [Reich, 1991]: design is an iterative process comprising several sub-tasks executed in sequence; and the diverse nature of these sub-tasks implies the integration of different knowledge types and reasoning capabilities. Design can be conceptualized as a five-task process: problem analysis, synthesis, analysis, redesign and evaluation (see figure 2.2).

The design process starts with the problem statement, which comprises the design problem specifications and requirements. The problem analysis consists on the elaboration of these specifications and requirements, trying to define them in a formal way. This task involves the use of domain knowledge to assist in the definition of the problem requirements. The result is the definition of the problem design space through the use of the specifications and requirements.

The next task consists in building the space of design candidates through a synthesis task. This task searches and explores the structural design space trying to identify candidate designs. This process is guided by control knowledge. The synthesis task can be interpreted as a transformation from intentional to extensional description of designs, using pairs of specifications and design descriptions to represent candidate designs.

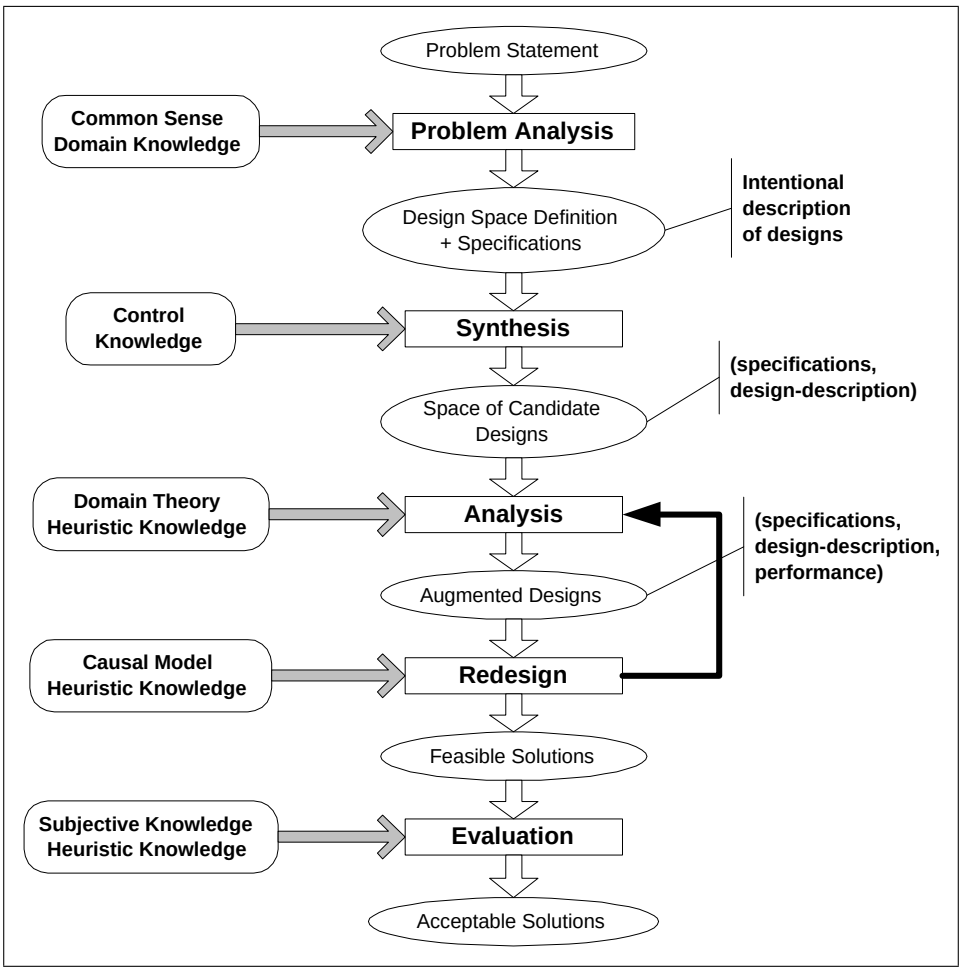


Figure 2.2: The tasks involved in the design process [Reich, 1991].

The analysis task has the goal of checking design constraints and assessing design performance. While the first part tries to identify design inconsistencies, errors or constraint violations, the assessment of the design performance is important to determine the quality of each candidate design. In the analysis task, the domain theory can be formalized in the form of heuristic knowledge (or other type of knowledge). The nature of analysis is very different from synthesis, while the synthesis task creates a design, the analysis task verifies and evaluates the created design.

Most of the times, analyzed designs are dropped because they are not feasible, or they need some redesign in order to comply with the design constraints. Solving design inadequacies is the goal of the redesign task. In this task the behavioral knowledge is very important, since it relates function with structure and it provides a way of knowing, which structural components to modify in order to achieve the

desired functionality. Causal models can be a way of modelling behavioral knowledge. Analysis and redesign can be an iterative process until the design complies with the desired requirements.

The final task involves the evaluation of the set of feasible designs, so that one of them (or more) can be implemented. Basically the evaluation task consists in ranking the designs that comply with the design constraints. There are several ways to evaluate designs and most of the times, this is done using subjective knowledge in the form of esthetics or simplicity criteria.

Though the tasks are shown in sequence, the design process can be more dynamic comprising several loops between the design tasks. This is an iterative process that converges to a solution under the strategic control of the designer.

The application of the CBR framework to the design process is another important decision within this work. The next subsection explores the main idea of CBR and the next subsections explore the issue of Case-Based Design.

2.1.2 Case-Based Reasoning

Case-Based Reasoning (CBR) [Kolodner, 1993, Maher et al., 1995] is the main reasoning framework used in our work. This framework allows the centralization of design knowledge and provides cognitive aid to designers. CBR can be viewed as a methodology for developing knowledge-based systems [Althoff, 2001] that makes use of experience for problem solving. The main idea is to reuse past experiences to solve new situations or problems, and possibly to learn from the system's reasoning.

A *case* is a central concept in CBR, and it represents a chunk of experience in a format that can be reused by a CBR system. Usually a case comprises three parts: problem, solution, and outcome.

The problem is a description of the situation that the case represents. It can be, for example, the symptoms of a patient in a medical case, or the requirements for a software system, or any description characterizing the situation that is being represented.

The solution describes what was used to solve the situation represented in the problem, or how the problem was solved. For instance, in a medical domain the

solution can represent the treatments prescribed to a patient. In the software domain a solution can be a design that complies with the system requirements.

The outcome expresses the result of the application of the solution to the problem. This means that, usually, there are two possible outcomes: success or failure. A success case represents a situation in which the solution worked well, while a failure case represents a situation where the solution did not work. A case can also comprise other parts like a justification, which can relate the problem features to the solution components through causal relations.

Another important part of a CBR system is the case library (or case base). Due to the high number of cases that a case library can have, most of the CBR systems use indexing structures that enable fast retrieval of relevant cases from memory.

At an abstract level CBR can be described by the reasoning cycle shown in figure 2.3 [Aamodt and Plaza, 1994]. The reasoning process starts with the problem description, which is then transformed into a target problem (or query problem and also called query case). The problem is provided by an user or by another system. The first phase in the CBR cycle is to retrieve from the case library the cases that are relevant for the target problem. The relevancy of a case must be defined by the system developer or the user, being the most common definition based on feature similarity. Retrieval returns the best case[1] and provides the input to the next phase along with the target problem.

The reuse phase (also designated as adaptation phase) modifies the retrieved case to fit the problem requirements, yielding a new case (or new solution). This process can make use of several inference techniques. Some work has been developed on the subject, but this is probably the less explored phase of the CBR cycle (see [Voss, 1996] for a description of several works on adaptation). The next step for a CBR system is to revise the new case, returning a tested and repaired case. This phase usually comprises two parts: verification and evaluation. While verification checks the new case consistency and coherence, the evaluation phase assesses the performance and characteristics of the new case. Finally, in the retain phase the new case is stored in the case library. This phase is more complex than it seems, because not all cases should be stored. If a new case is very similar to a case already in the library, then

[1]Depending on the system's goals and architecture, there are systems that retrieve more than one case.

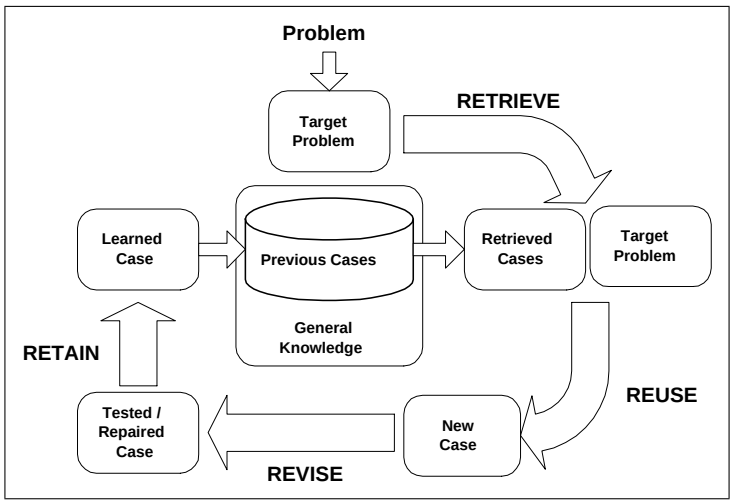


Figure 2.3: The CBR cycle adapted from [Aamodt and Plaza, 1994].

it should not be stored because it brings almost nothing new to the system. In fact, possibly, it will degrade the system performance. This last phase closes the CBR cycle by feeding the system with new experiences, making the system capable of learning. This reasoning cycle is more iterative in many CBR systems, with the possibility to backtrack to previous reasoning phases when something fails.

2.1.3 Case Representation

The application of CBR to design is called Case Based Design (CBD), and a parallel between CBR and design can be established (see table 2.1). The problem analysis is defined in CBR as the formalization of the target problem along with the associated indexes. The synthesis phase is equivalent to retrieval and reuse phases, despite some CBR systems do not have the reuse phase. The analysis and evaluation tasks are equivalent to the revise phase, both analyze designs or solutions. The redesign phase can be seen has the junction of the reuse and revise phases, depending on the type of verification mechanisms involved in the revision phase.

A CBD system uses cases as the main knowledge source, which is sometimes the only source. The case content determines the way the indexing memory is organized, which are the index types available, the reasoning processes, and other aspects of a CBD system. Case representation is also important to establish communication with

Table 2.1: Correspondences between design tasks and CBR phases.

Design Tasks	CBR Phases
Problem Analysis	Representation and Indexing
Synthesis	Retrieval and Reuse
Analysis	Revise
Redesign	Reuse and Revise
Evaluation	Revise

the designer. Cases must have information not only for the system's reasoning mechanisms, but also for the designers. As described in section 2.1.2, cases usually comprise three parts: problem, solution and outcome. In CBD the outcome is usually implicity in the design and a new part can be added to the case representation: the explanation or justification part. This is a description, usually in the form of causal knowledge, that associates problem specifications to structural components of the solution. The problem description is associated with the functional design space, comprising requirements and specifications. The solution is associated with the structural design space comprising the design structural components. The explanation description is associated with the behavior design space, and is usually formalized in the form of causal knowledge or domain models.

Some of the most common representation formalisms used in CBD are:

Attribute/Value Pairs In this representation cases are vectors of attribute/value pairs, in which an attribute is a design variable [Kolodner, 1993, chapters 5 and 6]. Design variables and the respective range of values must be defined. This representation is independent from the application domain and is one of the most frequent in CBR systems. Nevertheless, this representation has some limitations, especially in the expressive capabilities of the attributes.

Textual Representation Text is a natural way for humans to describe cases, and is used by some CBD systems [Goel et al., 1993, Maher et al., 1995]. Nevertheless, a textual representation is difficult to manipulate by the reasoning mechanisms due to the ambiguity normally involved in natural language. In order for the reasoning mechanisms to work properly, some level of natural language processing would have to be used. Despite these limitations, textual representations are designer-friendly.

Hierarchical Representation Using hierarchies is also a common way of representing cases in CBR systems [Smyth and Cunningham, 1992, Gomes and Bento, 1997a, Gomes and Bento, 1997b, Gomes and Bento, 1998]. Links of type *is-a* represent generalization relations between nodes, described by properties in the form of attribute/value pairs. These nodes can inherit properties from parent nodes. This is a powerful representation, highly structured and that benefits from some properties of taxonomical structures. Associated reasoning mechanisms are more complex and more prone to errors. Another type of hierarchical representation that instead of being a generalization taxonomy (with *is-a* relations) is a hierarchy in which links represent *part-of* relations.

Graph Representation This case representation is a complex representation that uses graph-like structures [Sycara and Navinchandra, 1991, Zhao, 1991, Duffy and MacCallum, 1989]. Nodes represent attributes or properties of the case, and links represent semantic relations between attributes. This representation is similar to semantic networks, and has the advantage of using the semantic relations as indexes. Reasoning algorithms that use this type of representation are more complex than the ones that use simple representations.

Multimedia Representation In multimedia representations, different types of information are combined [Domeshek and Kolodner, 1993]. This representation comprises text, sound, image and other types of information to represent a design. Though it is a very powerful representation technique, it needs two important functionalities: a tool capable of doing the integration of the information, and an adequate processing scheme. It is a representation oriented to the designer, which can be augmented with a more structured type of representation.

Other formalisms exist in CBD. Mainly these are formalisms that combine two or more of the basic formalisms described above.

2.1.4 Case Indexing and Storage

Case indexing and storage are tightly connected. The way cases are organized in memory, constraints the indexing schemes that can be used for retrieval. The main aspects that influence the case base organization are: case contents, paradigms used in indexing cases, and case visualization. Other two facts important in defining the case indexing and structure are: flexibility and efficiency. Flexibility because a case may have to be retrieved in different ways, depending on the retrieval perspective. Efficiency because the size of the case base can impose strong computational limitations.

Some of the most common indexing and storage mechanisms used in CBD are:

Flat Organization In a flat case base, cases are stored in a list (see [Kolodner, 1993, page 293]). This list can be organized in a specific order, like ranking cases alphabetically. But the main issue here, is that there is no clustering of cases to make the process faster. On the other hand, all the cases can be inspected and the retrieval accuracy can be greater comparatively to clustering structures. There is no indexing scheme, since cases are inspected one after another, starting at the beginning of the list. If the case base is large, the retrieval process can be impossible to perform due to time limitations. Adding new cases or deleting cases are trivial computationally inexpensive operations.

Hierarchical Organization One way to make the retrieval process more efficient, is to use a hierarchical structure to cluster cases. For example, decision trees or shared-feature networks (see [Kolodner, 1993, page 295]). In this approach, cases are indexed by taxonomy nodes, and can be accessed using these nodes as indexes. Generally, nodes are attributes, and edges have an associated value (a decision tree), which corresponds to indexing cases by attribute/value pairs. This organization works well with a attribute/value pair representation. Other ways to organize cases is by using part of cases as indexes, and to build a hierarchical partonomic structure that clusters the cases [Gomes and Bento, 1998]. An index in this situation would be a piece of a case.

Model Organization Another way to organize cases can be performed using models, like Structure-Behavior-Function models (SBF [Goel, 1992]), Function-Behavior-Structure models (FBS [Qian and Gero, 1996]), causal models or qualitative models (see CADET [Sycara and Navinchandra, 1991]), or even dependency network models. Each type of models has it's own specific way of indexing cases. We will focus on the SBF models cited above. The SBF models describe structure, behavior and function of the design they represent. Using the SBF formalism cases can be indexed using structural variables, behavioral variables, and/or functional variables. Some systems that use this representation formalism are capable of learning which are the relevant descriptors to be used in case indexing [Bhatta and Goel, 1992, Bhatta and Goel, 1993a, Bhatta and Goel, 1993b].

The indexing process consists on associating one or more indexes to a case, defining the situations in which the case is relevant. The index selection is an interpretation process based on the context in which the case is considered to be useful. There are three aspects to take into account in the indexing mechanism:

1. Indexing vocabulary.

2. Indexing process.

3. Index organization.

An index is a label associated with a case, which establishes a relation between the indexed case and the particular situation that the index represents. Indexes are usually associated with problem specifications and can have different levels of abstraction. An index that can be directly extracted from the case description is called a specific index. Abstract indexes are derived from the case description, but do not correspond directly to case features.

Specific indexes are easy to extract from cases, but are not in general good indexes, because they represent superficial aspects of the situation being described in the case. Abstract indexes are more difficult to extract, because extraction involves an inference step. In another way, they are more accurate for determining the case relevance.

The indexing vocabulary often coincides with the design attributes, or is a subset of these attributes. When abstract indexes are used, the vocabulary does not coincide with the description attributes.

Indexes must have certain characteristics that allow them to be effective. They must preclude the situations in which a case can be used. They must be abstract enough to be used in several situations, but specific enough to be easily identified. They must enable the discrimination between different scenarios, in such a way that the differences between a case and a target problem can easily be identified.

Case indexing can be performed manually by a designer, or in an automated way using indexing algorithms. There are three main ways to select indexes automatically: based on lists, based on differences, and based on explanations. In the list-based indexing there is a list of attributes that are used as indexes. The difference-based indexing uses differences between cases' descriptions to determine the relevant indexes. Indexing based on explanations extracts indexes using models of the working domain.

2.1.5 Retrieval

Retrieval of relevant cases is an important task for a CBD system. Case relevance can have different interpretations, depending on the system's goals. Case relevance is generally defined taking into account the similarity between a case description and the problem specification. Nevertheless, there are other ways of defining relevancy. For instance, in spite of being based on similarity with the problem, it can be defined in terms of the effort needed to adapt a case to the target problem (see [Smyth and Keane, 1995a]).

Retrieval can be decomposed in two tasks: retrieval of relevant cases from the case base (we will call it the retrieval task), and ranking of these cases against the target problem (the ranking task). In general, CBD systems incorporate both tasks, but there are systems that do not have one of these tasks.

The retrieval task is usually performed using the indexes, and is basically a search process in the indexing structure. Retrieval must be efficient and accurate in order to save time in the subsequent reasoning phases.

After a set of relevant cases are identified by the retrieval task, these cases are ranked in relation to the target problem. Usually cases are ranked using a similarity metric, which assesses the degree of similarity between the target problem and the case being ranked. A similarity metric can be a weighted sum of the degree of similarity between attributes.

An example of a similarity function is represented in equation 2.1 where, P is the target problem, C is the case, ω_i are the weights associated with the attributes $\left(\sum \omega_i = 1\right)$, $AttrSim$ is the attribute similarity, p_i is a problem attribute, c_i is a case attribute, and n is the number of attributes in the case description).

$$Similarity(P, C) = \sum_{i=1}^{n} \frac{\omega_i \cdot AttrSim(p_i, c_i)}{n} \tag{2.1}$$

Weights are a way of assigning different importances to attributes. There are other similarity metrics, like recursive metrics for hierarchical case representations [Bergmann and Stahl, 1998], or less sophisticated metrics, like counting the number of word occurrences in a textual description of a case.

An important aspect for retrieval is attribute matching. A simple way to perform attribute matching when cases are represented by attribute/value pairs, is just to match attributes with the same name. But even in this rather simple approach there can be situations in which it fails. An example of this, is when it is allowed the same attribute to occur two or more times in a case description. The matching phase would have to decide which attributes to match. More elaborated schemes use semantic similarity of attributes, or in hierarchical or graph case representations structural matching, which introduces a great deal of complexity into the similarity metric.

There are some systems that make no distinction between retrieval and ranking, since they use a K-Nearest Neighbor (KNN) algorithm [Dudani, 1976] to retrieve and rank cases. This algorithm applies the similarity metric described in equation 2.1 to all cases in the case base. In this situation, there is no retrieval task in the sense of using indexes to retrieve cases from the case base. The algorithm selects the K most similar cases to be retrieved. This can be a time consuming way of doing retrieval, but one of the most accurate ones, because it performs a global search in the case base.

2.1.6 Adaptation

After the retrieval phase, the selected cases may have considerable differences against the target problem, making necessary for the CBD system to adapt one or more retrieved cases to fit the target problem description [Kolodner, 1993, pages 393-437]. This phase is named adaptation and comprises the modification of one or more retrieved cases to address the specifications of the target problem. There are three basic adaptation methods (see [Kolodner, 1993, page 395] or [Maher et al., 1995, pages 110-111]): substitution, transformation and derivational replay.

Substitution methods replace values or components of the retrieved case. In a transformation method, rules or procedures are used to modify values or case components. Analogy replay assumes that the retrieved case includes the method or procedure used to generate the case solution. Using this generation method, derivational replay applies it to the target problem to generate a new solution that fits the target problem.

Most of the adaptation methods use domain knowledge to perform modifications in the case solution. There are several types of domain knowledge used by adaptation methods. The most used ones are: heuristics, domain models, causal models, rules, and semantic nets. In the remaining of this subsection we will detail some of the most common adaptation methods.

Substitution methods work by replacing a case element that is not coherent with the specification for the target problem. One possible substitution method is reinstantiation. It assumes that the retrieved case solution fits the target problem. Based on this, it uses the same structure from the case's solution to build a new solution, with old components replaced by ones with the same structural and functional roles, but applicable to the target problem specifications. Another substitution method is parameter adjustment, which uses interpolation to replace a variable's value. This method is normally used on numeric variables, though it can also be applied to symbolic attributes. Local search is a method used for replacing a solution substructure. This method is local to the old solution, and it uses other memory structures to select the elements that are going to be replaced. Another method in this category is case-based substitution, which uses other cases to find a substitute component.

In the substitution methods it is assumed that there is a component in the old

solution that can be replaced. In transformation this to-be-replaced component does not exist, thus it is necessary to transform the retrieved solution so that it fits the target problem. There are several transformation methods. Two of them are the use of rules and the use of domain models. In the rule-based transformation the adaptation is determined by a set of rules that transform the retrieved solution. In the model-based transformation, the retrieved solution is modified using domain models. Another form of performing transformation adaptation is by using heuristics to determine what should be modified in the old solution. These heuristics are highly dependent on the application domain.

The previous methods modify the retrieved solution to reach a new one. In derivational replay it is not the retrieved solution that is reused, but the process that originated the retrieved solution. What this method does, is to replay the generation process used for construction of the retrieved solution, but now using the target problem specifications. The outcome will be a new solution that fits the target problem specifications.

2.1.7 Verification and Evaluation

After generating a new solution, the CBD system should check the design constraints and possibly evaluate the design properties. The first task is called verification and consists on the assessment of the design correctness, looking for design constraints that are violated or incoherent design components. The next task is called evaluation and it comprises the assessment of the design properties and performance.

If the verification phase detects errors in the design, then the system can perform three types of response. It can try to correct the errors, using methods similar to those used in adaptation, like domain models or causal models; it can return the solution to the adaption phase, so that adaptation mechanisms can try to eliminate the design flaws; or the verification module can ask the designer to correct the errors, followed by learning of adaptation knowledge.

There are two main verification approaches: model-based and case-based. The first uses domain models to assess the design correctness, while the second uses knowledge in cases to try to detect design errors.

The evaluation task can have two different goals: to inform the designer about the design properties and performance; or to select designs based on their properties and performance. The first goal has an informative value, and provides information to the designer on the design's quality. The second goal possesses a selective value, which can be used to choose the generated designs that will be presented to the designer.

Two types of aspects that can be evaluated in a design are design properties and design performance. The properties concern the static characteristics of a design, like how many components it has, or how they are linked together, while the performance evaluates the dynamic aspects of the design, focusing on how well the design achieves the intended functionalities. An obstacle is that some design aspects are difficult to evaluate in an automated way due to their subjectivity. When dealing with this kind of subjective aspects the designer has an important role in evaluating the design.

2.1.8 Learning

The reasoning phase that closes the CBR cycle is the retain phase (or learning). This is the phase when new cases can be added to the case base, supplying the CBD system with new knowledge. This phase relates directly to Case Base Maintenance (CBM), which is the way the case base is managed. There are many issues are related with CBM, some of them are: should a case be added to the case base or not? Which are the indexes that a new case should have? Which is the limit of the case base size? What should be the case base contents? Several works in this area address these questions but few apply to case representations different from attribute/value pairs. Complex representations are more difficult to deal with, than attribute/value pair representations and a lot of work has still to be performed in this topic.

An important concept that relates to those difficulties is subsumption. Case subsumption [Racine and Yang, 1997] defines that a case is redundant if: is equal to another case, is equivalent to another case, or is subsumed by another case. This is an important definition, which is used in several CBM works to define redundancy in case bases. A redundant case should not be added to the case base, since it will degrade the response time of the system.

Another important concept in CBM is case coverage [Smyth and McKenna, 1998].

Case coverage can be defined as the portion of the design space that a CBD is able to cover, be it with the retrieval mechanisms alone or also with the adaptation mechanisms. The capability of evaluating the case base in terms of coverage is important to determine if a case is redundant or not.

These are just two of the main concepts used in CBM, for more details about previous works see also sections 4.6 and 6.7.

2.2 Creativity and Design

This section starts by describing an important concept in this dissertation: Creativity. Then we focus on a specific area of creativity, which is Creative Design, exploring this concept from a cognitive and computational point of view. Finally this section ends with an overview of the analogical reasoning process from a creative design perspective.

2.2.1 Creativity

Creativity is defined as a cognitive process that generates a product that satisfies specific properties [Dasgupta, 1994], like usefulness or novelty. Creativity has at least four components [Brown, 1989]: creative process, creative product, creative person or entity, and creative situation. Within this model a crucial component is the creative product, and an important issue in developing computational models of creativity, is the evaluation of the creative product. Research on this domain came up with several methods used for generation of artifacts considered creative [Partridge and Rowe, 1994, Gero, 1994a, Gero and Maher, 1993, Pereira and Cardoso, 2001, Gomes et al., 1996]. Some methods identified in creative reasoning are: cross-domain transfer of knowledge (analogy), combination of knowledge, and exploration and transformation of conceptual spaces. Most of these methods are used in creative design [Gero, 1994b].

Margaret Boden [Boden, 1990] introduced the concepts of Personal-Creativity (P-Creativity) and Historical-Creativity (H-Creativity), in order to distinguish personal creativity from social creativity. She labels an idea or artefact as P-Creative, when

the creator classifies it as creative. H-Creativity is defined as being a social phenomenon, where a social group is responsible for the creative classification, but not for the creation of the idea or artefact. While H-Creativity concerns to a social phenomenon, P-Creativity is a personal experience. This dichotomy between P-Creativity and H-creativity worths the comments and criticism of the scientific community that studies creativity and creative systems [Turner, 1995, Perkins, 1995, Ram et al., 1995, Schank and Foster, 1995, Wiggins, 2001]. Some works point out other dimensions in creativity that are not mentioned in Boden's classification [Pease, 2001, Colton et al., 2001].

Evaluation of creativity implies the confrontation of two different knowledge bases. One comprises the knowledge within the product being evaluated (the knowledge that the product encodes or transmits). The other is the evaluator's knowledge, which is taken as a reference. As said before, the evaluator can be the creator or a group of entities (society). In the first case, if the product is defined as creative, it is also P-Creative, because the evaluator and the creator are the same entity. When the society labels the product as creative, it is also H-Creative.

H-Creativity is related to ideas that are created by the first time by a society or by a civilization. Using this definition, the concept of H-Creativity is directly related to the evaluation scope. As an example, we can consider the invention of powder in European countries after it was discovered in China. This was surely an H-Creative product for the Western society, but not for Oriental people. H-Creativity's scope is the society or a civilization's culture (any group of people big enough to be considered a social group).

This type of creativity has a society or culture as a reference, which makes quite difficult or inadequate to have a computer as the evaluator agent. Social knowledge is constantly being modified, maintaining a degree of incoherence and uncertainty. These are aspects that a computer tool has problems to deal with, in case it would take the goal of evaluating H-Creativity. It is our opinion that the better a computer can do is to evaluate the product in the role of one individual or entity, but not as a society. These reasons take us to think that H-Creativity evaluation is out of scope from computer systems, which left us with P-Creativity evaluation as a task for artificial systems.

Why use a computer to explore creativity? Can a computer help us formalize the main principles of human creativity? Can computers be creative? Can computers recognize creative designs? These are questions that some Artificial Intelligence and Cognitive Science researchers try to answer. For the reasons presented before, we can only try to answer positively these questions when reporting to P-Creativity.

In contrast to H-Creativity, P-Creativity occurs when its creator classifies a product as being creative. In this way the creator and evaluator are the same entity. Suppose a math student discovers a new property of prime numbers, when trying to solve a math problem. S/he then goes to the math teacher to tell her/his discovery, but the math teacher says that a Portuguese mathematician from the thirteenth century had already discovered it. From the student's point of view, it is a P-Creative discovery, although it has long been discovered before, which hinders it from being H-Creative.

Although recognizing that P-Creativity involves evaluation by the creator, this is not enough. One must also take into account the generative processes that originated the creative product. Boden states that a P-Creative idea must not be produced by the same generative rules as other familiar ideas. This implies that a genuinely creative idea, in a P-Creative sense, must have been produced by a novel process or by a conceptual space transformation, which is a more radical change. A computer program can be seen as a "set of rules", which at a first glance take us to think that a computer can not be P-Creative. But, what about a computer system that learns and has the capability of changing its own conceptual space through external interaction? If we compare with a human being, we conclude that this is also a way to explain human creativity.

The next subsection explores the combination of creativity and design, in what is called creative design.

2.2.2 Creative Design

Creative design can be defined as a cognitive process that generates new classes of designs. During the design process, the designer explores the space finding new pieces of knowledge enabling the creation of new design classes. The Creative Process

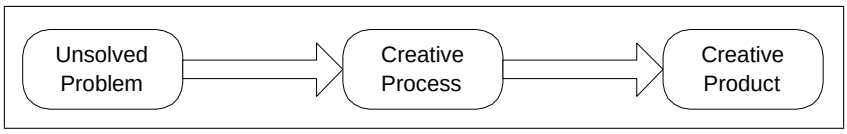


Figure 2.4: The creative design path.

generates new designs, the outcome is the Creative Product (see figure 2.4).

As enumerated before, some properties are commonly associated with a product labelled as creative [Dasgupta, 1994]. These properties determine the product's creativity, thus defining guidelines to the product's evaluation. In the remaining of this section we present some core properties that characterize a creative design from two different perspectives: the creative process and the creative product.

Processes for routine design are likely to be different from those for creative design [Gero and Maher, 1993]. This raises an important issue: does creative design involve intention from the point of view of the designer? This is a controversial question that the research community has not answered yet. For instance, can a certain degree of randomness be regarded as a characteristic of a creative process? Researchers working on evolutionary systems can say that it has its role in the process. But can chance be regarded in the same way? Can a person come up with a creative solution to a problem, even if there was no intention to solve the problem?

We will focus on more tangible aspects of the creative process having in mind that the processes for creative design are mainly different from those for routine design, though we think that some of the reasoning processes applied in creative design are used in routine design and vice versa.

One consequence of the definition of creative design is that possible solutions cannot be pre-established, at least, explicitly. Otherwise it would be routine design. But the frontier between explicit and implicit definition of the possible solutions is difficult to determine. Possible solutions are defined by the knowledge that is available. If the space of solutions does not comprise all the possible solutions it must be incomplete, sometimes being also contradictory. These are two characteristics with which the creative processes must deal. Creative processes work with knowledge comprising distinct characteristics from those found in routine design. Creative processes must also comply with knowledge needed to perform cross-domain transfer of

ideas, which is regarded as one of the types of reasoning associated to creative design [Sycara and Navinchandra, 1991].

Design specifications consist on constraints and functional specifications. Both define the space of possible problems, thus constraining the space of possible designs. In creative design these spaces change during problem solving. Two possible space modifications are addition and substitution [Gero, 1994b]. Addition consists in integrating new design specifications to the problem space or new designs to the solution space. Substitution comprises the replacement of a region of a design space by another set of design specifications or solutions. This can be achieved by replacing design variables.

Reasoning processes modify the search space. During this modification the reasoning processes generate new design solutions. The generation of several design alternatives is important for exploration of the space of possible solutions and the space of possible problems, thus assisting the designer in understanding the problem nature.

Two important processes in the creative process are: search and exploration (these were defined in section 2.1.1). Search is generally related with routine design and is defined as a computational process that requires a good definition of the search space [Gero, 1994b]. Exploration is a computational process that modifies the search space and is commonly associated with creative design. Flexible search mechanisms are necessary for creative reasoning [Gero and Maher, 1993]. These mechanisms are responsible for the exploration of the design space.

Some of the main reasoning processes involved in creative design are: mutation, combination, analogy, reasoning from first principles and emergence [Gero, 1994a]. Mutation comprises the modification of a design structure in order to generate a new one. Another way for generation of new solutions is the combination of pieces from different designs. This process can work at different levels of the design - functional, behavioral or structural level. Analogy is regarded as one of the more important processes in creative design. It comprises mapping between a source design and a target design. It is a suitable mechanism for transfer of ideas across different domains, thus implementing cross-domain fertilization. Reasoning from first principles makes use of domain models in order to generate new designs. These models are causal

or qualitative and can be viewed as "from scratch" generators of new regions in the search spaces. Emergence is a process in which additional design attributes are identified besides the intentional ones. This reasoning mechanism concerns to the ability to view things in new ways, which is a characteristic of creative reasoning [Partridge and Rowe, 1994].

Kolodner and Wills [Kolodner and Wills, 1993] claim that indexing cases accordingly to various perspectives is also important for case-based creative design. Kolodner and Wills propose the use of exploration processes that can search the case memory for cases that might be represented in a different way from the current problem representation. More recently Simina and Kolodner [Simina and Kolodner, 1997] propose a computational model that accounts for opportunistic behavior, as a characteristic of creative behavior in case-based design.

Sycara [Sycara and Navinchandra, 1991] produced also important work in this area. They propose a thematic abstraction hierarchy of influences as a retrieval method for case-based creative design. In this framework case organization provides the main mechanism for cross-contextual reminding which is very important in non-routine design.

Bhatta and Goel [Bhatta and Goel, 1997a, Bhatta and Goel, 1997b] proposed an analogical theory for creativity in design. This theory is based on generic teleological mechanisms, which are behavior-function models. These models are abstracted from the case-specific structure-behavior-function models and are used for complex topological modifications in designs.

Concerning to the creative product, the first thing that comes into mind is novelty, which is in our opinion a mandatory characteristic. The creative product must be something different from what the evaluator knows or thinks about. As explained before, evaluation of a creative product has to do with the confrontation of two sets of knowledge. One is the knowledge contained in the product and the other is the knowledge that the evaluator possesses or uses in the evaluation. If the knowledge from the product does not make part of the evaluator knowledge set, then it can be said that the product is novel.

During the evaluation of the presumably creative product, the evaluator uses performance-measuring functions, in order to determine to which extent the product

satisfies the problem requirements. These measuring functions make part of the evaluator's body of knowledge. If minimal performance thresholds are met, then the product is useful and is considered to solve the problem. This is another mandatory characteristic of creative solutions. The creative product must be appropriate and useful.

At an historical level, creative solutions must be novel in which regards to the society knowledge (H-Creativity). In this way, evaluation of creative solutions comprises a social dimension. This makes the automatic evaluation of creative products quite difficult and complex, leading to a two-step process. The first step is called internal validation and consists on checking that the proposed design comprises all the intended requirements (checking for usefulness). This phase is also called verification and most times can be performed automatically. The second step evaluates the novelty of the design and is normally performed by the evaluator/designer. It comprises the evaluation of the solution in regard to the information possessed by the judge. This phase is called validation.

Another aspect concerns to the situation in which the designer and the evaluator are the same entity. When these two entities are the same computational system, a possible benefit is the automation of the process, thus making it more efficient. Another advantage is the guidance that the evaluator can provide during the generation process. Nevertheless the computer system can hardly have the same interpretation of the problem as a human evaluator.

Also, it is necessary to establish a distinction between two types of systems in creative design. One is the kind of systems that support creative design, which are commonly designated as creative design support systems. Another type of systems automate the complete creative design process, leaving no role for the designer. While in a creative design support system the creative design results from the collaboration between the designer and the system, in an automatic system the designer only defines the problem, leaving all the generation work for the computer. These two types of systems have quite different purposes and philosophies concerning the roles of the user and the system, and also the creative design. Notwithstanding, the reasoning mechanisms and programming paradigms can be shared by both kinds of systems.

In both systems another important aspect is the user interface. This plays a strong

role on a support system but is also important in an automatic system. The user interface is relevant in the sense that it enables the system to communicate with the designer.

Another important aspect of a creative design system is that it must be able to store and use huge amounts of knowledge[2]. This will not only enable cross-domain analogy in the solution generation phase, but also will be helpful in the verification phase, when the system must determine if the retrieved information or generated solution is relevant to the problem itself. As said before, design uses functional, behavioral and structural knowledge. A creative design system will have to be able to use and represent these different types of knowledge. As described before, functional knowledge is commonly associated with the design problem, structural knowledge with the design solution and behavior with the explanation of how the design structure accomplishes the intended functions. Control and search mechanisms in a creative design system are also very important. These must be more powerful and flexible comparatively to other knowledge-based systems, in order to deal with the complexity associated with creative design.

2.2.3 Analogical Reasoning in Creative Design

As mentioned before, analogical reasoning is a widely used problem solving method in design [Gentner, 1983, Hall, 1989, Holyoak and Thagard, 1989]. It consists on the transference of knowledge from one source domain to a different domain (usually designated as target domain). This transference is based on similarities between a past situation and the current one, which are then used to transfer knowledge, generating solutions for the current situation or problem. The main benefit of analogical reasoning is its capability to transfer knowledge between domains. This cross-domain transfer enables the generation of new designs, and can be considered a creative process [Gero, 1994a].

Creative design processes work with knowledge comprising distinct characteristics from those found in routine design and also comply with knowledge needed to perform

[2]This is a statement made by Roger Schank at his invited talk in the first International Conference on Case-Based Reasoning in Sesimbra, Portugal, 1995.

cross-domain transfer of ideas, which is regarded as one of the types of reasoning associated with creative design [Sycara and Navinchandra, 1991]. It comprises mapping a source design and a target design, and we use this mapping process to implement cross-domain fertilization.

Analogical reasoning enables the exploration of new areas of the solution space, because it maps concepts from the source domain into a target domain. Since software reuse can be seen as a transfer of knowledge between a source project into a target one, analogy can be a suitable mechanism for software reuse. Suggestions made by the system to the designer using analogical reasoning can also stimulate new ideas in the designer. Analogical reasoning can also work in cooperation with the software designer, providing new ideas that the designer evaluates and selects.

The software design level is a suitable level for analogical reasoning, because it deals with concepts, enabling the system to reason on a more abstract level, one in which it can easily switch between different domains.

Hall [Hall, 1989] defines a descriptive framework for computational analogy in four steps:

Recognition: identification of analogous sources using a target problem as a search probe.

Elaboration: mapping of target and source, with the possibility of involving deeper inferences.

Evaluation: evaluation, repair, justification and possible extension of the mappings and inferences previously established.

Consolidation: learning of the new knowledge resulting from the mapping and inferences performed during the previous phases.

These steps have a parallel with CBR, which from our point of view contributes to an easy integration of analogy and CBR.

2.3 Software Design and Reuse

This section describes the application domain of this thesis: software design and reuse. We start by describing what is software design, and then provide an overview on the area of software reuse. This section also describes the idea of software design patterns and the Unified Modelling Language (UML), which are both used in our work.

2.3.1 Software Design

Software development usually comprises five steps: analysis, design, coding, testing, and integration [Boehm, 1988] (the V-model). In our work we focus on the design step. This is justified by several factors: it is an important phase (most of the decisions made at this stage have a great influence on other phases); it is a task more complex than the analysis phase, requiring more expertise and know-how from the developers; there are few computational tools providing cognitive support to designers; and knowledge in this phase is at a more abstract level and less formal, than in the coding phase. These are some of the reasons that make the design phase a crucial part in the software development process and a challenging problem.

The design phase can be divided into two sub phases: conceptual design (or general design) and detailed design. In the conceptual design phase, the solution is analyzed, defining the types of entities and subsystems that comprise the design model. This is a high-level task, in which the main concerns are related with the specification phase and not with the implementation phase. The result is a conceptual design that identifies the software architecture, so that it is able to satisfy the specification produced in the analysis phase. The goal of the detailed design sub phase is to prepare the software's coding step. The output is the definition of the data structures and algorithms for the coding phase. The detailed design model serves as a high-level view of the source code, describing how the code should be organized and which are its key features.

2.3.2 Software Design Reuse

Software reuse [Coulange, 1997] has been a hope in the software development community since it started. The main expectation was that reuse would enable software development to be faster, better and cheaper.

The most common form of software reuse is code reuse, which is not always the more efficient one. Any type of knowledge generated during software development can be reused, from system requirements to test documents. We are interested in the reuse of design knowledge in the form of diagrams or other representation formalisms.

If adequately applied, software reuse must provide several advantages to software development:

- Productivity should increase with reuse, because reusing a component[3] must be cheaper than developing it from scratch.
- If a component is reliable, it must remain so when it is reused. Before components can be reused, they benefit from debug and testing, and if the component is latter revised in order to improve it, a future system using it will benefit from this improvement.

Nevertheless, the successful reuse of software implies also that some requirements must be met:

- All extensions of a component must be able to be used without special effort by the software that reuses the component.
- A reusable component must be able to be assembled without difficulty, with other reusable components.
- A reusable component must be suitable for adaptation to a new context without affecting it's other users.

Some of the problems associated with software reuse are: finding, understanding, modifying and composing components. Finding a component is a task similar to retrieval in CBR. The goal is to find the most similar design in the design repository

[3]We use the term component here to refer to any type of software knowledge that can be reused.

(in CBR it would be the case base). After finding the most similar component, this may not be similar enough to be reused without modifications. It is important to develop ways to modify components. Composition of components is a way of modifying them. Composition is more complex than the previous phases, requiring a good level of understanding of the component. The problem of understanding components is a common issue to all the other three aspects involved in reuse, because it provides the basis for reasoning with components.

In the last decade some forms of design reuse have emerged in the software engineering community, from software design patterns [Gamma et al., 1995], to frameworks [Johnson, 1997], and component-based development [Atkinson et al., 2002]. These forms of reuse seem much more promising than the usual way of code reuse (despite being also forms of reusing code), and they are starting to gain importance in the research community.

The first form of reuse ever used was copying [Coulange, 1997, page 8]. The system developer identifies the component to be reused, copies it to a new location and adapts it to the new context. Although being a very simple procedure it has problems. It is not efficient, and if it takes place without any kind of computational support, it can be impracticable, due to the size of the repository.

Component-based development [Atkinson et al., 2002] is a form of software reuse, where components are individual and independent units. This makes components highly reusable, creating ready-to-assemble and ready-to-run units. Using component-based development is more an assembly task than a programming process. A common problem of this method, is to find an adequate component for reuse. Component-based development promotes "reuse in the small", since components are the smallest coherent units of software that make sense as stand alone reusable entities.

Design patterns [Gamma et al., 1995] provides high level reuse. Functioning at an architectural design level, it addresses reuse at a higher level than component-based reuse. Software design patterns describe ways of solving abstract software design problems. They can be viewed as providing guidelines for assembling entities in the form of classes, interfaces or other component types. Because our work also involves design patterns, section 2.3.3 describes them in more detail.

Frameworks [Johnson, 1997] are an object-oriented reuse technique, and can be viewed as a form of design reuse. A possible definition for framework is: a reusable design of all or part of a system that is represented by a set of abstract classes. This set of abstract classes needs to be customized by the application developer in order to build an application. Frameworks are at a different level of abstraction in relation to design patterns, which can be interpreted as micro-architectural elements of frameworks. If component reuse is "reuse in the small", frameworks are "reuse in the big".

The application of reuse techniques might not be enough to improve software design and consequently software development. There are two main approaches in design reuse [Prieto-Diaz and Jones, 1988, Prieto-Diaz, 1993]: reuse of designs and design for reuse. The first approach uses already developed components and prepares them for later reuse by storing them in a design repository. The second approach takes the opposite way. When developing a system, the development team must think in terms of making components reusable, that is, they should plan for the future. There are pros and cons in both approaches. The first approach is more simple at the beginning, and it provides a way of taking advantage of already developed and tested components. Although, these components may not be in the best format or abstraction level to be reused, constraining their reuse scope. The design for reuse approach involves a greater commitment from the the development team, which must take development time to think ahead and foresee future usages of the components. But this work may be compensated in the future. The main effect of design for reuse is that it changes the policy for software development within a company. It must be part of the software development methodology.

2.3.3 Software Design Patterns

A software design pattern describes a solution for an abstract design problem. This solution is described in terms of communicating objects[4] that are customized to solve the design problem in a specific context. A pattern description comprises four main elements:

[4]The term object can have several meanings in computer science, in this document we will use the term *object* to refer to classes, interfaces and packages, and not as in the object-oriented programming paradigm, in which an object is an instance of a class.

Name: Identifies the design pattern, and is essential for communication between designers.

Problem: Describes the application conditions of the design pattern and the problem situation that the pattern intends to solve. It also describes the application context through examples or object structures.

Solution: Describes the design elements that comprise the design solution, along with the relationships, responsibilities and collaborations. This is done at an abstract level, since a design pattern can be applied to different situations.

Outcome: Describes the consequences of the pattern application. Most of the times patterns present trade-offs to the designer, which need to be analyzed.

Gamma et al. [Gamma et al., 1995] describe a catalog of 23 design patterns, and give a more detailed description for each pattern consisting on: pattern name and classification, pattern intent, other well-known names for the pattern, motivation, applicability, structure, participants, collaborations, consequences, implementation example, sample code, known uses, and related patterns. From these items, we draw attention to the participants and the structure. The participants describe the objects that participate in the pattern, along with their responsibilities and roles. These objects play an important role in our approach. The structure is a graphical representation of the design pattern, where objects and relations between them are represented.

A Pattern is classified based on its function or goal. The pattern categories are: creational, structural, and behavioral. Creational patterns have the main goal of object creation, structural patterns deal with structural changes, and behavioral patterns deal with the way objects relate with each other, and the way they distribute responsibility.

As an example of a design pattern we briefly present the Abstract Factory design pattern (see [Gamma et al., 1995], page 87). The intent of this pattern is to provide an interface for creation of families of objects without specifying their concrete classes. Basically there are two dimensions in objects: object types, and object families. Concerning the type of objects, each type represents a group of objects having the same conceptual classification, like window or scrollbar. The family of objects defines

a group of objects that belong to a specific conceptual family, not the same class of objects. For example, Motif objects and MS Window objects, where Motif objects can be windows, scrollbars or buttons, which exist in MS Window objects but do not have the same visual characteristics.

Suppose now, that an user interface toolkit is being implemented. This toolkit provides several types of interface objects, like windows, scroll bars, buttons, and text boxes. The toolkit can support also different look-and-feel standards, for example, Motif, MS Windows, and Mac OS. In order for the toolkit to be portable, object creation must be flexible and can not be hard coded. A solution to the flexible creation of objects depending on the look-and-feel, can be obtained through the application of the Abstract Factory design pattern. This pattern has five types of participating objects:

- The **Abstract Factory** object declares an interface for operations that create abstract products.
- The **Concrete Factory** objects implement the operations to create concrete products.
- The **Abstract Product** objects declare an interface for a type of product object.
- The **Concrete Product** objects define a product object to be created by the corresponding concrete factory, and also implement the Abstract Product interface.
- The **Client** objects use only interfaces declared by Abstract Factory and Abstract Product classes.

A possible solution structure for the problem posed by the interface toolkit is depicted in figure 2.5. The pattern participants are: abstract factory (*WidgetFactory*), concrete factories (*MSWindowsFactory* and *MotifFactory*), abstract products (*Window* and *ScrollBar*), concrete products (*MSWindowsWindow*, *MotifWindow*, *MSWindowsScrollBar*, and *MotifScrollBar*), and client (*Client*). The create methods in the factories are the only way that clients can create the interface objects, thus controlling and abstracting object creation. The main consequences of this pattern are that

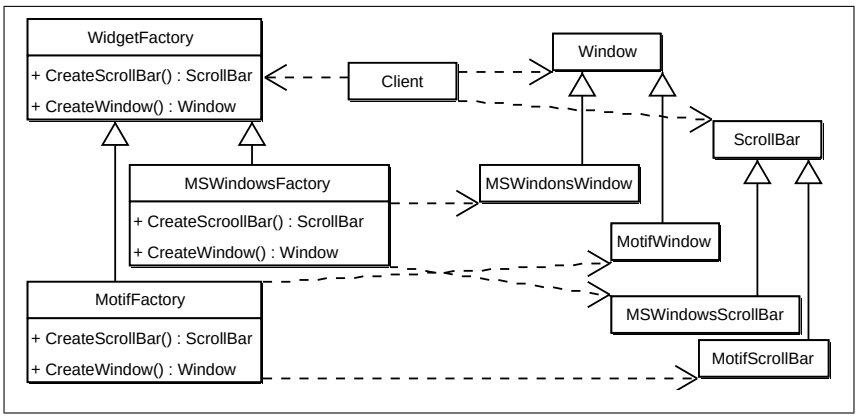


Figure 2.5: The application of the Abstract Factory design pattern to the interface toolkit problem.

it isolates concrete classes, makes exchanging product families easy, and promotes consistency among products.

2.3.4 Unified Modelling Language

The Unified Modelling Language (UML) [Rumbaugh et al., 1998] is a graphical notation to represent both system requirements and design. UML comprises a set of diagrams that can be used to model a software system. These diagrams are: use case diagrams, class diagrams, sequence diagrams, collaboration diagrams, state diagrams, activity diagrams, deployment diagrams, and component diagrams.

Use cases are snapshots of the system being modelled. They can be interpreted as a set of sequence steps describing an interaction between an external entity (a user or another system) and the system. These steps have in common a user goal. An use case diagram is a diagram for visualization of use cases. These diagrams comprise actors, use cases and relations between these entities. An actor is a role that an external entity plays with respect to the system. Actors can be humans or other computer systems, basically they can be any entity capable of communicating with the system.

Class diagrams describe the structure of a software model by describing what are the object types in the system and what kind of relationships exist between them (see figure 2.6). So a class diagram represents two types of elements: entities and relations between entities.

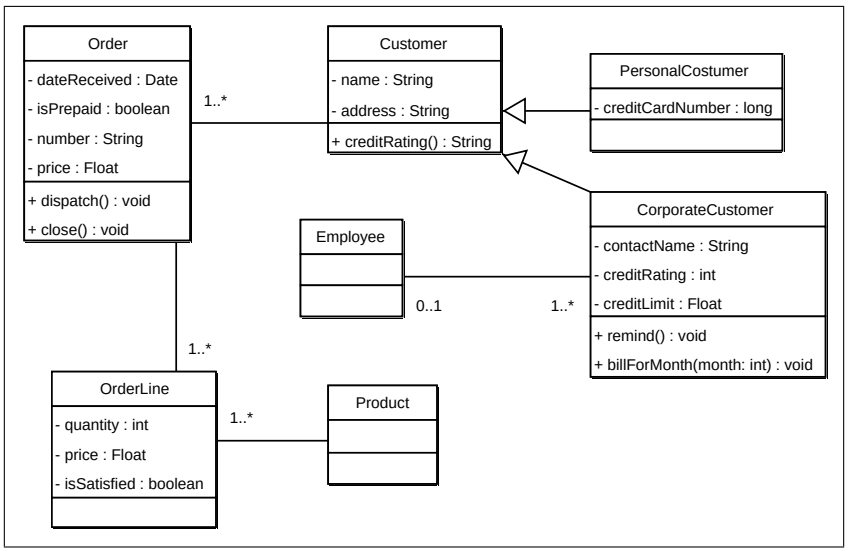


Figure 2.6: An example of a class diagram.

There are three types of entities in a class diagram: classes, interfaces and packages. A class describes an entity in UML and it corresponds to a concept described by attributes at a structural level, and by methods at a behavioral level. Attributes represent class properties and possess a data type, a scope and a default value. Methods are processes that the class can perform. They describe the behavioral aspect of the class. Methods have a scope, a name, a parameter list, and a return type expression. Interfaces have only method declarations, since they describe a protocol of communication for a specific class. A package is an UML object used to group other diagram elements.

Links represent relationships between UML objects. There are four types of relations: generalizations, dependencies, realizations and associations. A generalization establishes an *is-a* relation between two classes. The main consequence is that the subclass inherits all the attributes and methods from the parent class (or classes in case of multiple inheritance). Dependencies represent a relation between two elements, in which a change to the definition of one element may influence changes in the other element. A realization is a relation between a class and an interface, in which the interface defines a specific protocol of communication for the respective class. Associations represent conceptual relationships between classes. The association is a catch-all relation, and can represent almost any kind of semantic relation. It

comprises an aspect that the former three relations do not address, that is multiplicity. A relation between two UML elements can have multiplicities attached to each relation element. Multiplicity denotes the number of objects that may participate in the association.

Three levels of class diagram perspectives can be defined: conceptual, specification and implementation. They represent the system being modelled from a more abstract perspective (conceptual) to a more specific one (implementation). In the conceptual perspective the class diagram represents the main entities in the system. This is a software independent view of the system model and entities defined on the diagram may not have a code counterpart. The specification level describes the system model taking into account the interactions between model entities and interactions between the system and external systems. The implementation perspective is intended to be a conceptual representation of the implementation and it deals with implementation issues and details. The three perspectives are important for a good system modelization and all three have to deal with design issues. These perspectives are a variant of the two parts of the design phase in software development, discussed in subsection 2.3.1.

A sequence diagram tries to capture the behavior of a use case. It represents several example objects and messages that are passed between them. One of the main aspects of these diagrams is the time sequence of events, which defines how the system should behave in a particular situation (or use case).

Collaboration diagrams have the same purpose as sequence diagrams but represent interaction between objects in a different way. Objects are represented by boxes and arrows indicate messages being passed between objects, but on this type of diagram, sequence is indicated by numbering the messages.

The goal of state diagrams is to describe the system's behavior. This is achieved by describing all the possible states of a particular object, and how state changes occur. The most common use of these kind of diagram is to show how an object behaves during its lifetime. It also defines how the object's state changes in response to external messages from other objects. Nodes in this type of diagram, represent the object states and state transitions are represented by links between nodes. Transitions can have three types of labels: event, conditions and action. The semantics associated

are: the transition is valid if the event occurs, and the conditions are met, then the action is performed.

Activity diagrams are a good way of describing parallel processing. They represent the sequencing of activities, and can also represent conditional and parallel behavior. The main element of an activity diagram is the activity state (or activity), which represents a processing state. This processing can be a real-world process or an execution of a software routine. There are several elements in an activity diagram. Activities are represented by rounded boxes. Arrows represent transitions between diagram elements, which can have conditions associated to them. There are elements that are used for parallelizing the control flow. These elements are forks and joints, which are represented by horizontal black bars. There is a start and an end node, represented by black circles. Finally there are branch and merge elements that allow the representation of conditional control. They are shown as diamonds.

Deployment and component diagrams are used to show the physical relationships among software and hardware units. Nodes represent computational units, most of the times hardware, and are depicted by 3-D boxes. Components are depicted by boxes inside nodes, and they represent a code module. There are two types of relations: connections and dependencies. Connections represent the communication links between hardware nodes. Dependencies relate components and have a similar semantic to class diagram dependencies.

Chapter 3

A Case-Based Approach for Reuse in Software Design

This chapter describes REBUILDER, the system that implements our approach. REBUILDER has two main goals: create a corporative memory of software designs, and provide the software designer with a design environment able to promote reuse of software designs. This is achieved with CBR as the main reasoning mechanism. REBUILDER comprises four modules: Knowledge Base (KB), UML Editor, KB Manager and CBR engine (see Figure 3.1). REBUILDER interacts with two different types of users: software designers, and KB administrators. Software designers view REBUILDER as a CASE tool, and as a system with reuse capabilities. A KB administrator has the role of updating and maintaining the consistency of the KB.

REBUILDER is based on a client-server architecture comprising two servers and two types of clients (see Figure 3.2). REBUILDER's KB comprises a WordNet server and a file server, while the clients comprise the administrator and the designer modules. The main difference between the two types of clients is that the manager client has an extra module allowing KB maintenance, the rest is the UML editor. There can only be one server of each type, and only one manager client. There can be several designer clients depending on hardware resources.

The UML editor is the front-end of REBUILDER and provides the environment for the software designer. Apart from the usual editor commands to manipulate UML objects, the editor integrates new commands able to reuse design knowledge. These commands correspond to the CBR capabilities provided by REBUILDER.

The KB manager module is used by the administrator to keep the KB consistent

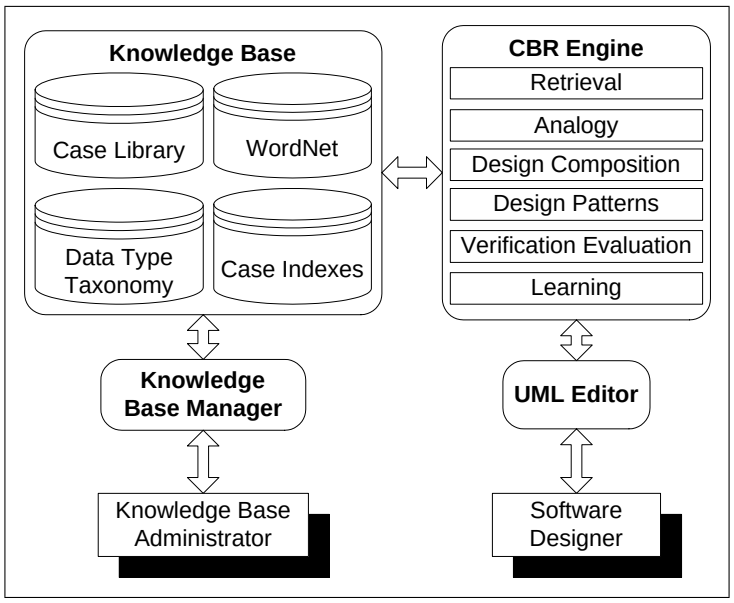


Figure 3.1: REBUILDER's Architecture.

and updated. This module comprises all the case base management functions implemented in REBUILDER. These are used by the KB administrator to update and modify the KB.

BEBUILDER's KB comprises four different parts: case library, which stores the cases of previous software designs; index memory, used for efficient case retrieval; data type taxonomy, which is an ontology of the data types used by the system; and WordNet, which is a general purpose ontology [Miller et al., 1990]. The KB is described ahead in this chapter in more detail.

The CBR Engine is the reasoning module of REBUILDER. As the name indicates, it uses the CBR paradigm as a reasoning framework. This module comprises six different parts: Retrieval, Analogy, Design Composition, Design Patterns, Verification, and Learning. All these modules are detailed in the following chapters.

3.1 UML Editor

For the UML editor we used an open source editor called ArgoUML[1]. Figure 3.3 presents a screen shot of the editor. Basically, it comprises four main areas. The title

[1]The web page of ArgoUML is http://argouml.tigris.org/.

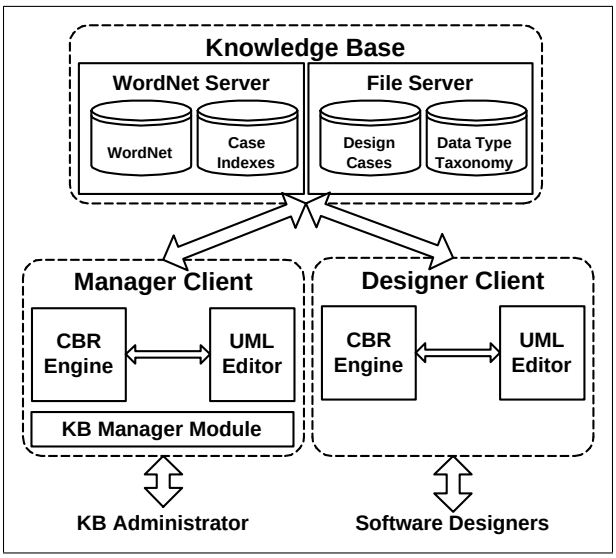


Figure 3.2: REBUILDER's Architecture from an implementation point of view.

bar and the command menu, which comprise the normal commands for windows-based applications. REBUILDER commands were inserted in the command menu. At the left side of the command menu, it presents a navigation tree, where the designer can navigate on the model tree. The designer can also select which type of view s/he wants, for instance, package-view or diagram-view (the combo box on top of the navigation tree). The third part is the central right area, where the diagrams are displayed. It also comprises a tool bar with some command buttons. The four buttons on the right are REBUILDER commands: Retrieval, Analogy, Design Composition and Design Patterns. The fourth region is the bottom area, which comprises several tabs. These tabs concern to: properties, documentation, style, source, and constraints of UML objects.

REBUILDER's UML editor comprises commands for manipulation of UML objects, for design knowledge reuse and management of the KB contents. These commands are organized into two main categories:

- **Knowledge Base actions:**
  - Create new knowledge base.
  - Open knowledge base.
  - Close knowledge base.
  - Manage the case library.

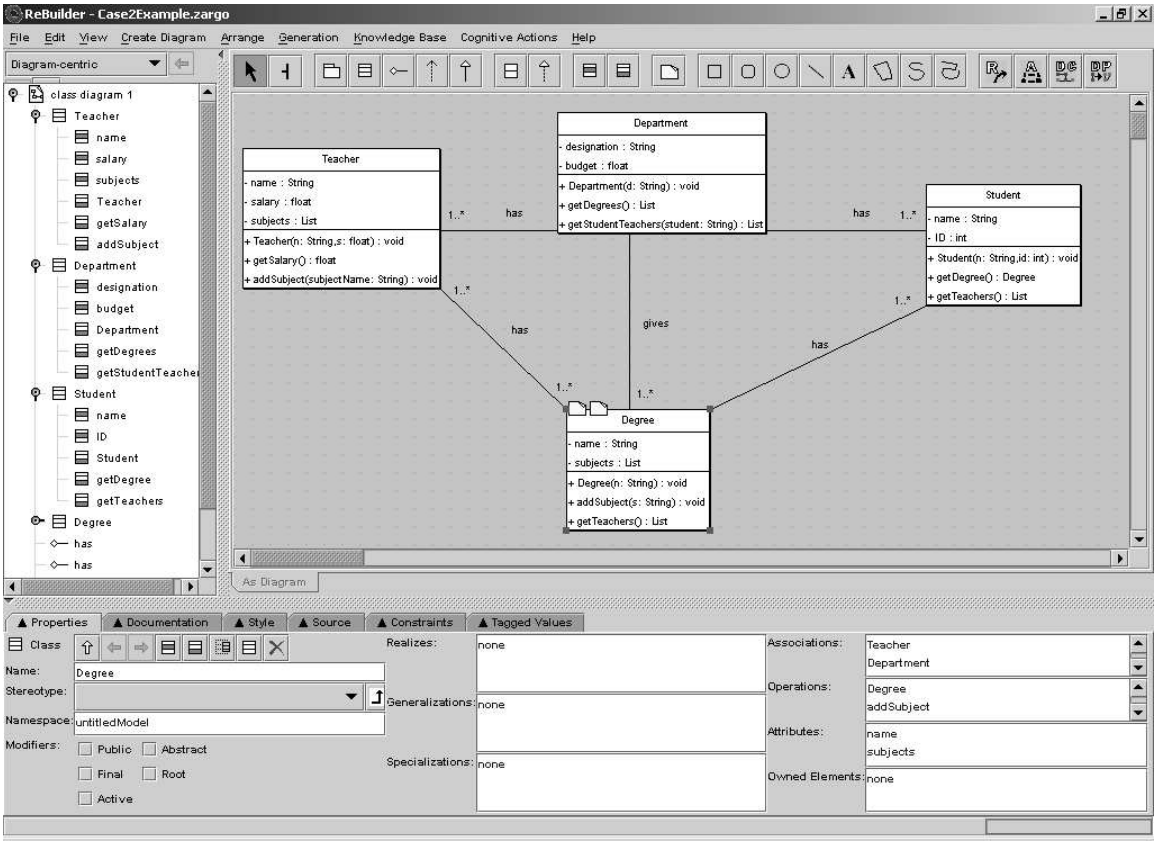


Figure 3.3: UML editor of REBUILDER.

- Manage the DPA[2] case library.

- **Cognitive actions:**
  - Retrieve designs or objects.
  - Reuse designs using analogy.
  - Reuse designs using design composition.
  - Reuse designs using software design patterns.
  - Verify designs.
  - Evaluate designs using object-oriented metrics.
  - Activate learning strategies for case base maintenance.
  - Submit designs to the case library.
  - Manage classification of design objects.

[2]Design Pattern Application, described in detail in section 3.3.1.

– Modify settings of cognitive actions.

3.2 Knowledge Base Manager

The KB Manager is used by the administrator to manage the KB, keeping it consistent and updated. This module comprises all the functionalities of the UML editor, and adds case base management functionalities. These are used by the KB administrator to update and modify the KB. The list of available functions are:

KB Commands: Create, open and close a KB.

Case Library Manager: Open Case Library Manager, which comprises functionalities to manipulate the cases in the case library, like adding new cases, removing cases, or changing the status of a case (see section 3.3.1 for case status).

Activate Learning: Gives the KB administrator an analysis about the contents of the case library. REBUILDER uses several case base maintenance techniques (see section 4.6) to determine which cases should be added or removed from the case library.

Settings: Adds extra configuration settings which are not present in the normal UML Editor version used by the software designers. It also enables the KB administrator to configure the reasoning mechanisms.

Through the KB manager the administrator supervises the knowledge in the system. The basic responsibilities of the administrator are to setup the system, to decide which cases should be in the case library, and to revise new diagrams submitted to the case library by software designers.

Deciding the contents of the case library is not an easy task, especially if the case base has a large number of cases. In this situation the KB manager module provides a set of case-based maintenance policies that the administrator can run to determine which are the cases that should be removed or added to the case library.

When a diagram is submitted by a software designer as a candidate for a new case to be added to the case library, the administrator has to check some items in the

diagram. First the diagram must have synsets associated to the classes, interfaces and packages. This is essential for the diagram to be transformed into a case, and to be indexed and reused by the system. Diagram consistency and coherence must also be checked. REBUILDER provides a verification and evaluation mechanism that can help the administrator in two different ways: verification can identify certain syntax and semantic errors or less common items in the diagram an draw the administrator's attention to them; evaluation assesses the design properties and characteristics, presenting them as guidelines for the diagram's evaluation, this is performed using several object-oriented metrics [Rosenberg and Hyatt, 1996, Bär et al., 1999]. Finally the administrator can use the case-based maintenance policies to help him decide if the submitted diagram should be added to the case library.

3.3 Knowledge Base

The KB stores all the knowledge used by the reasoning mechanisms. It comprises a case library, the WordNet ontology, case indexes and the data type taxonomy. Each of these parts is described in this section.

3.3.1 Case Library

The case library comprises two types of cases: design cases and design pattern application cases (DPA cases). The design cases are the most frequently used in REBUILDER, and represent UML class diagrams. DPA cases represent specific situations of design pattern applications. In the text, we use the term case to refer to a design case, if we want to refer to a DPA case, we will explicitly use the DPA acronym to distinguish it.

Cases are stored in files - a file for each case - and they are only read to memory if necessary. Cases in a case library are organized in three lists. A list of confirmed cases, comprising the cases used for reasoning that were approved by the KB administrator. The list of unconfirmed cases, containing the cases submitted to the case library by designers. These cases are not used for reasoning and wait for revision by the KB administrator. They can be accepted to be included in the case base, moving to the

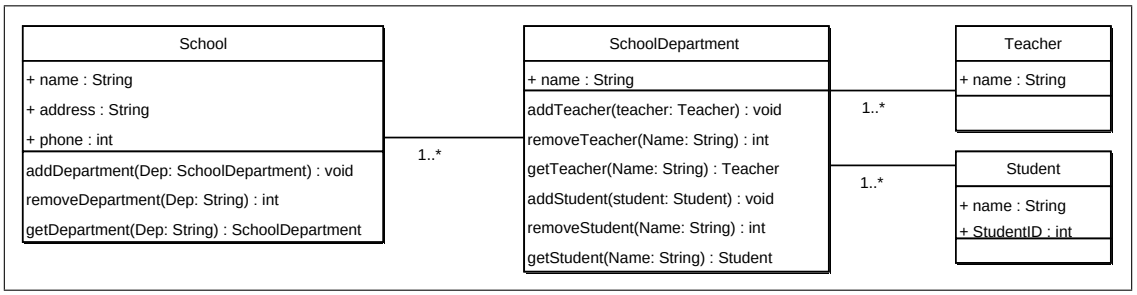


Figure 3.4: Example of an UML Class diagram (*Case1*), the package classification is *School*.

confirmed list or, if rejected, they can be deleted or moved to the obsolete list. This list comprises cases that were considered obsolete for various reasons.

The next subsections describe the representations for design cases and DPA cases.

Design Cases

In REBUILDER a case describes a software design, which is represented in the UML formalism through the use of Class Diagrams. Figure 3.4 shows an example of a class diagram representing part of an educational information system. Nodes represent classes and comprise a name, attributes and methods. Links represent relations between classes. Conceptually a case in REBUILDER comprises: a name used to identify the case within the case library; the main package, which is an object that comprises all the objects that describe the main class diagram; and the file name where the case is stored. Cases are stored using XML/XMI (eXtended Mark-up Language, XML Metadata Interchange), which is a widely used format for data exchange. UML class diagram objects available in REBUILDER are: packages, classes and interfaces. See section 2.3.4 for more details.

DPA Cases

A DPA case describes a specific situation where a software design pattern was applied to a class diagram. Each DPA case comprises: a problem and a solution description. The problem describes the application situation based on: the initial class diagram, and the mapped participants. The initial class diagram is the UML class diagram to which the software design pattern was applied. This is the diagram before the application of the design pattern. The mapped participants are specific elements that must be present in order for the software design pattern to be applicable. Participants

can be: classes, interfaces, methods or attributes. Each participant has a specific role in the design pattern and it is determinant for the correct application of the design pattern. Each pattern has it's specific set of participants. Once the participants are identified, the application of a design pattern follows a specific algorithm that embeds the pattern actions. To select the role for each participant in the initial class diagram, a mapping of these participants is performed.

It is important to describe the types of participants defined within our approach. Object participants can be classes or interfaces, attribute participants correspond to class attributes, and method participants correspond to object methods. Each participant has a set of properties:

Role (String): Role of the participant in the design pattern.

Object (class or interface): Object playing the role, or in case of attribute or method participant the object to which the attribute or method belongs.

Method (method): Method playing the role in case of a method participant.

Attribute (attribute): Attribute playing the role in case of an attribute participant.

Mandatory (Boolean): True if the participant must exist in order for the design pattern to be applicable, if it is optional then the value is false.

Unique (Boolean): True if there can be one or more participants with this role type, otherwise it is false.

The solution description of a DPA case is the applied name of the design pattern, which is then used to select the correspondent software design pattern operator (described in detail in subsetion 4.4.2). Different DPA cases can have the same solution, because what a DPA case represents is the context of application of a design pattern, and a large number of context situations is possible.

3.3.2 WordNet

WordNet is used in REBUILDER as an ontology. It uses a differential theory where concept meanings are represented by symbols that enable a theorist to distinguish

among them. Symbols are words, and concept meanings are called synsets. A synset is a concept represented by one or more words. Words that can be used to represent the same synset are called synonyms. A word with more than one meaning is called a polysemous word. For instance, the word mouse has two meanings, it can denote a rat, or it can express a computer mouse. In this way the word mouse belongs to more than one synset.

WordNet is built around the concept of synset. Basically it comprises a list of word synsets, and different semantic relations between synsets. The first part is a list of words. Each word is linked to a list of synsets that the word can represent. The second part, is a set of semantic relations between synsets. REBUILDER uses four semantic relations: *is-a*, *part-of*, *substance-of*, and *member-of*. Synsets are classified into four different types: nouns, verbs, adjectives, and adverbs.

Synsets are used in REBUILDER for categorization of software objects. Each object has a synset associated, which is the synset that was selected as the correct synset accordingly to the specific object diagram. In order to find the correct synset, REBUILDER uses the object name, and the names of the objects related with it, which define the object context. The object's synset can then be used for computing object similarity (using the WordNet semantic relations), or it can be used as a case index, allowing rapid access to objects with the same classification. WordNet is also used to compute the semantic distance between synsets. This distance is the length of the shortest path between the two synsets. Any of the four relation types can be used to establish the path between synsets. This distance is used in REBUILDER to assess the type of similarity between objects, and to choose the correct synset when the object's name has more than one synset. This process is called name disambiguation [Ide and Veronis, 1998] and is a crucial process in REBUILDER. If a diagram object has a name with several synsets, then more information about this object has to be gathered to find which synset is the correct one. The diagram objects that directly or indirectly are associated with this object are used for this disambiguation. In case the object is a class, its attributes can also be used in the disambiguation process. This procedure is used when a case is inserted in the case library or when the designer calls the retrieval module, section 4.7 describes this in more detail.

One problem that we encountered in WordNet, was that it had few concepts

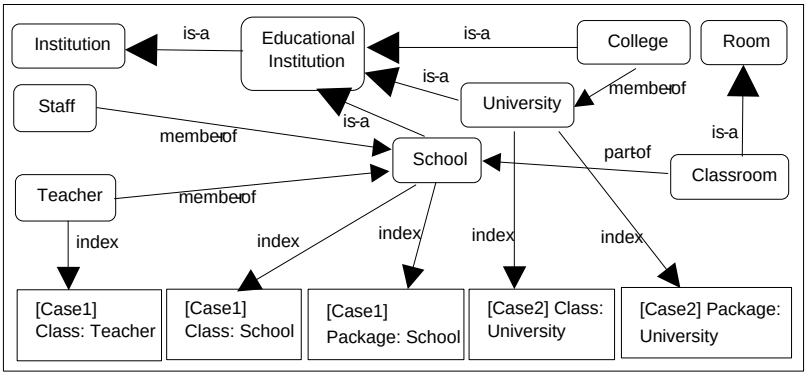


Figure 3.5: A small example of the WordNet structure and case indexes.

concerning specific knowledge on the software development domain. So we decided to integrate the concepts of the Java Class Hierarchy. This is the hierarchy of classes of the Java language. We connected this hierarchy to the WordNet structure by establishing an *is-a* link from the synset representing the concept of *computer program* to the Java class *object* (the top of the Java class hierarchy).

3.3.3 Case Indexes

Cases can not be stored in memory due to their dimensions, so they must be stored in files, which makes case access slower then if they were in main memory. To solve this problem we use case indexing. This provides a way to access the relevant case parts for retrieval without having to read all the case files from disk. Each object in a case is used as an index. REBUILDER uses the synset of each object to index the case in WordNet. This way, REBUILDER can retrieve a complete case, using the case root package, or it can retrieve only a subset of case objects, using the objects' indexes. This allows REBUILDER to provide the designer the possibility to retrieve not only packages, but also classes and interfaces. To illustrate this approach, suppose that the class diagram of figure 3.4 represents *Case1* and figure 3.5 presents part of the WordNet structure with the case indexes associated with *Case1*. As can be seen, WordNet relations are of the types *is-a*, *part-of* and *member-of*, while the *index* relation links a case object (squared boxes) with a WordNet synset (rounded boxes). For instance *Case1* has one package named *School* (the one presented in figure 3.4), which is indexed by synset *School*. It has also a class with the same name and categorization, indexed by the same synset, making also this class available for retrieval. Another advantage of indexing cases by their parts (diagram objects) is that

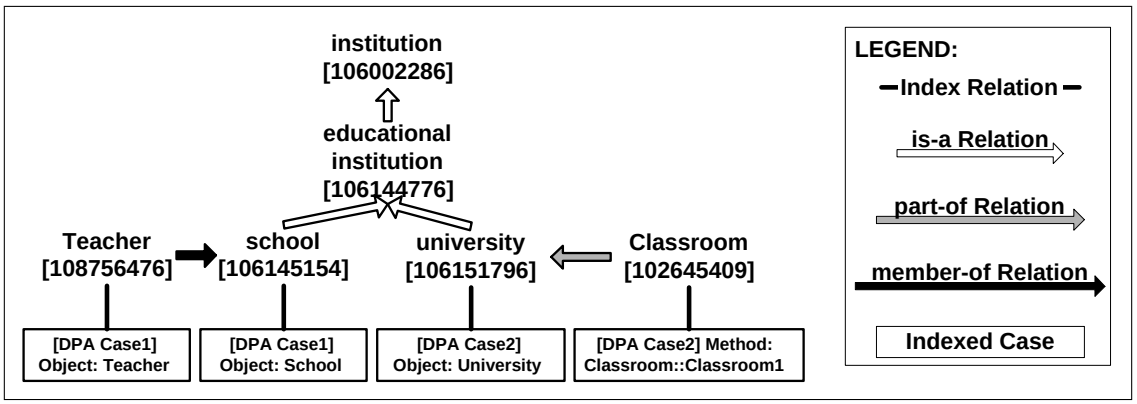


Figure 3.6: An example of the DPA case indexing. Synsets are identified by nine digit numbers.

cases can be retrieved in other situations, in which a part of a case can be relevant. The indexing of *Case1* by the synset *Teacher* illustrates this.

DPA cases are indexed using the synsets of the object participants (figure 3.6 presents an example) and only the participants (objects, attributes and methods) can be used as retrieval indexes. The WordNet structure is used as an index structure enabling the search for DPA cases in an incremental way. Each case can be stored in a file, which may be read when it is necessary . In figure 3.6 there are four indexed objects, three of them corresponding to object participants (*Teacher*, *School* and *University*), and one to a method participant (*Classroom1*), indexed by the object comprising the method.

3.3.4 Data Type Taxonomy

The data type taxonomy is a hierarchy of data types used in REBUILDER. Data types are used in the definition of attributes and parameters. The data taxonomy is used to compute the conceptual distance between two data types. Figure 3.7 presents part of the data type taxonomy used in REBUILDER.

3.4 CBR Engine

The cognitive functionalities of REBUILDER are supported by the CBR engine. It comprises six submodules, each one implementing a different cognitive process. These

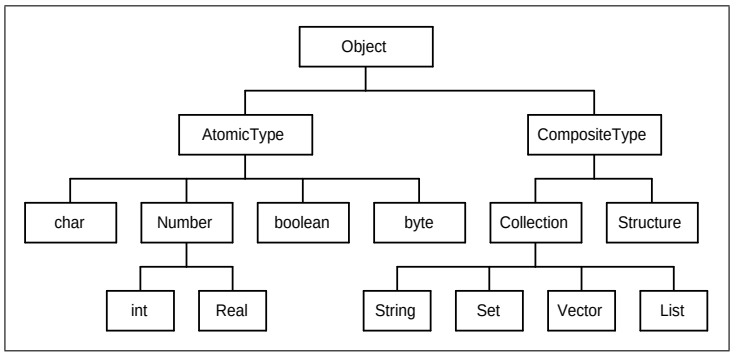


Figure 3.7: Part of the data taxonomy used in REBUILDER.

cognitive processes can be combined in several ways. Figure 3.8 presents a possible sequence of use for these cognitive processes. The CBR engine submodules are:

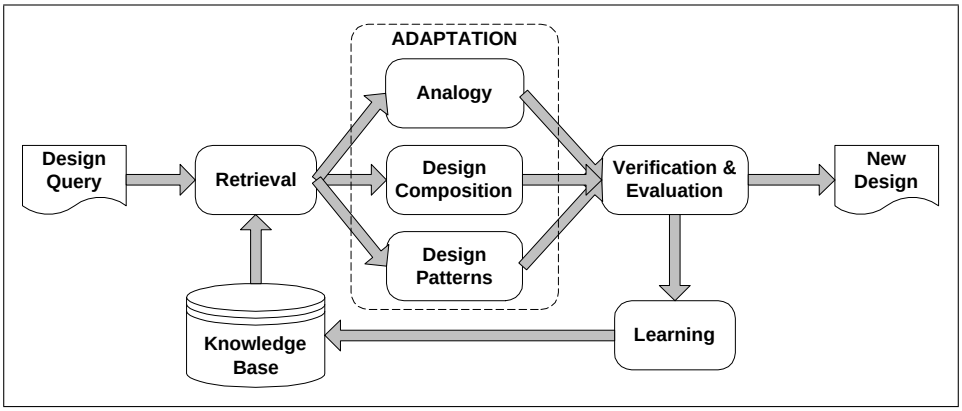


Figure 3.8: One possible combination of the CBR engine modules.

Retrieval: The retrieval submodule selects from the case base a set of cases ranked by similarity with the designer's query. It enables the software designer to browse through the most similar designs in the case library, exploring different design alternatives and reusing cases. This module works like an intelligent search assistance, which starts retrieving the relevant cases from the case library and then ranks them. The cases are ranked before being presented to the designer. Figure 3.9 presents an illustration of the retrieval process, which is detailed in section 4.1.

Analogy: The analogy submodule generates new solutions using analogical reasoning. This involves selecting a candidate from the case library, then mapping it with the query diagram, and finally transferring knowledge from this source case to the problem diagram, yielding a new diagram (see figure 3.10). This

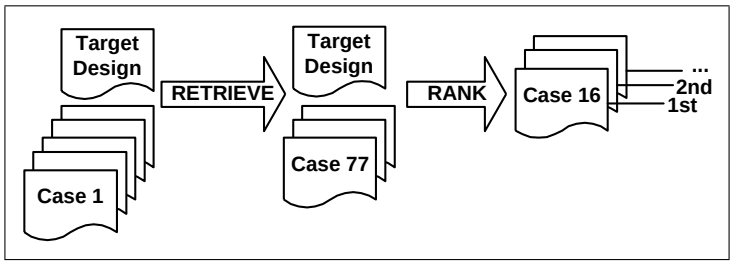


Figure 3.9: The retrieval process in REBUILDER.

mechanism generates solutions using only one case, which constraints the type of designs that it can generate. See section 4.2 for a detailed description.

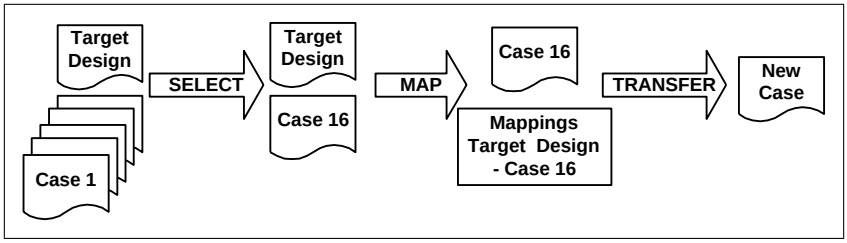


Figure 3.10: An illustration of the analogical reasoning process in REBUILDER.

Design Composition: The design composition submodule also generates new solutions from cases in the case library (see figure 3.11). The main difference to analogy generated solutions is that it can use more than one case to generate a solution. This mechanism can select pieces of cases and then compose them in a new diagram, yielding a solution to the designer's query. See section 4.3 for details.

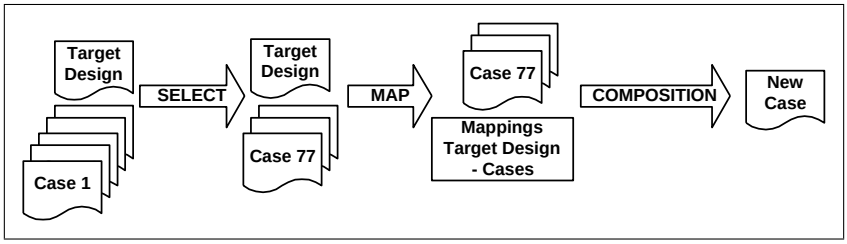


Figure 3.11: The design composition process in REBUILDER.

Design Patterns: The design patterns module can be used to generate new class diagrams based on the application of software design patterns (see figure 3.12). It is used a CBR approach to select which design pattern to apply, and how it should be applied. Alternatively, the user can select a design pattern to be applied to a diagram. The application is then performed automatically by REBUILDER through the use of pattern operators. REBUILDER acquires new DPA cases from the user interaction, see section 4.4 for a complete description.

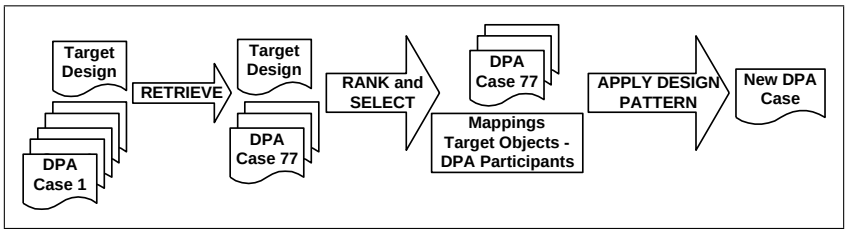


Figure 3.12: A generic view of the design pattern module functioning in REBUILDER.

Verification and Evaluation: This module comprises two functionalities: verification and evaluation. While verification checks the design coherence and correctness (see figure 3.13), the evaluation mechanism is used to assess the design's properties. The verification can be used in combination with analogy or design composition to look for errors in the generated solution and to correct them. The evaluation mechanism is at the designer's disposal to judge the design properties, trying to identify shortcomings in the design. Both processes are presented in more detail in section 4.5.

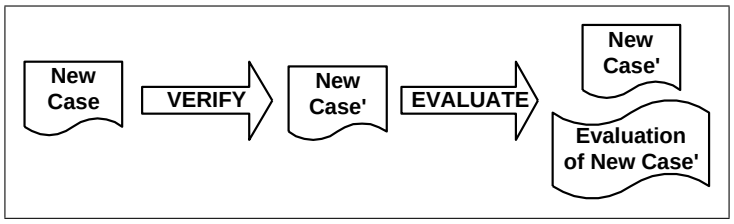


Figure 3.13: The verification process in REBUILDER.

Learning: The learning submodule implements several case-based maintenance strategies that can be used by the KB administrator to manage the case library. This module presents several measures to assess the case library performance, which provide an important advice to the administrator regarding the addition or deletion of cases (see figure 3.14). These strategies will be detailed in section 4.6.

There is also another module, which is the word sense disambiguation module, which is used when REBUILDER needs to associate a synset with a software object (see figure 3.15). This module receives a software object and a diagram, and returns a synset for the given object. The selection is based on several word sense disambiguation methods, which are described in section 4.7.

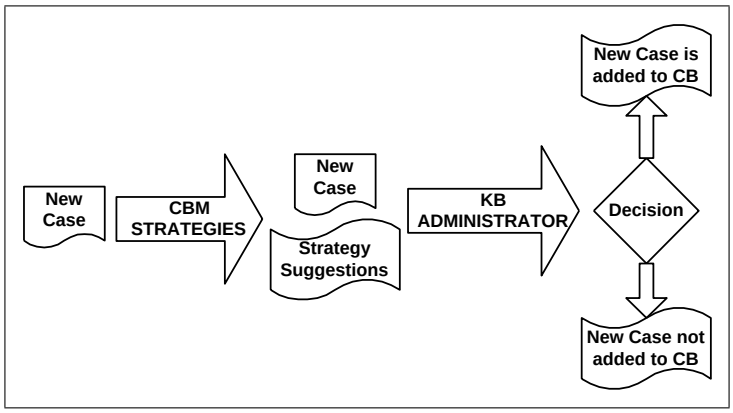


Figure 3.14: The process of case base maintenance in REBUILDER.

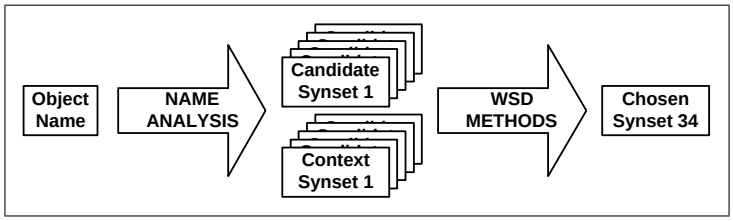


Figure 3.15: The word sense disambiguation process in REBUILDER.

Chapter 4

CBR Engine

This chapter describes the various reasoning modules of the CBR engine. Each module has a specific function in the reasoning cycle of CBR (see figure 3.8), though some of them come from different reasoning paradigms. The CBR methodology provided the framework for an integration of these various reasoning mechanisms. There is a module that is transversal to the CBR cycle, and it is not shown in REBUILDER's architecture. It is the word sense disambiguation (WSD) module, which has the function of associating a synset to a software object. This module can be used at any phase of the process, depending on the reasoning needs.

We start this chapter describing the retrieval module, ending with an example, which illustrates the functioning of this module. The second section describes the analogical reasoning module and how new diagrams are generated using analogy. This section also ends up with an example. Section three describes the design composition module, which is an alternative to the analogy module. Section four describes our CBR approach to the application of software design patterns, and presents an example of how this module can help design evolution. The next section describes the verification and evaluation module. Section six describes the case base maintenance (CBM) strategies used in the case learning module, and finally section seven concludes this chapter presenting the word sense disambiguation (WSD) process and the various disambiguation methods implemented in REBUILDER.

4.1 Retrieval Module

The case retrieval module can retrieve three types of objects: packages, classes or interfaces, depending on the object selected when the retrieval command is chosen by the designer. If the selected object is a package, the associated diagram will be considered as the query diagram (also designated as target design or problem).

The next subsections describe the retrieval algorithm, the similarity measures, the matching algorithms and the independence measures used in this module. The retrieval algorithm is used to retrieve a first set of cases, which are then ranked using the similarity metrics. The matching algorithms are used by the similarity metrics to select which objects to match. The independence measures are used by several reasoning mechanisms in REBUILDER, in particular, the matching algorithms use the independence measures to rank the lists of objects by element importance in the UML class diagram. At the end of this section we present a retrieval example to illustrate the retrieval process.

4.1.1 Retrieval Algorithm

The retrieval algorithm is the same for all three types of objects (packages, classes and interfaces), and is based on the object classification using the WordNet synsets.

Suppose that the N best objects have to be retrieved, *QObj* is the query object, and *ObjectList* is the universe of objects that can be retrieved (usually *ObjectList*[1] comprises all the objects in the library cases). The algorithm is described in figure 4.1, where MSL is the maximum search level allowed in WordNet.

The algorithm starts with the list of found objects ($ObjsFound$) empty, gets the synset for *QObj*, initializes the list of synsets to be explored ($PSynsets$) with the synset for *QObj* and initializes the list of synsets explored ($SynsetsExplored$). Then the algorithm cycles through a sequence of steps while the number of found objects is less than N, and there are synsets to be used as probes in the search, and the number of iterations is less than the maximum search level that is allowed (MSL). The steps inside the while cycle are: get the first element of $PSynsets$ as the current

[1]This list is called *ObjectList* because it can contain three types of objects: packages, classes and interfaces.

1. $ObjsFound \leftarrow \varnothing$
2. $PSynset \leftarrow$ Get synset of $QObj$ (WSD process)
3. $PSynsets \leftarrow \{PSynset\}$
4. $SynsetsExplored \leftarrow \varnothing$
5. WHILE ($\#ObsFound < N$) AND ($PSynsets \neq \varnothing$) AND ($IterationNumber < MSL$) DO
6. $Synset \leftarrow$ Remove first element of $PSynsets$
7. $SynsetsExplored \leftarrow SynsetsExplored + Synset$
8. $SubSynsets \leftarrow$ Get $Synset$ hyponyms (children nodes)
9. $SuperSynsets \leftarrow$ Get $Synset$ hypernyms (parent nodes)
10. $SubSynsets \leftarrow SubSynsets - SynsetsExplored - PSynsets$
11. $SuperSynsets \leftarrow SuperSynsets - SynsetsExplored - PSynsets$
12. $PSynsets \leftarrow$ Add $SubSynsets$ to the end of $PSynsets$
13. $PSynsets \leftarrow$ Add $SuperSynsets$ to the end of $PSynsets$
14. $Objects \leftarrow$ Get all objects indexed by $Synset$ and with the same type as $QObj$
15. $Objects \leftarrow Objects \cap ObjectList$
16. $ObjsFound \leftarrow ObjsFound \cup Objects$
17. ENDWHILE
18. $ObjsFound \leftarrow$ Rank $ObjsFound$ by similarity
19. RETURN the first N elements from $ObjsFound$

Figure 4.1: The retrieval algorithm used in REBUILDER.

probe synset; add this synset (*Synset*) to *SynsetsExplored*; get the parents and children synsets of *Synset*; intersect these synsets with the lists *SynsetsExplored* and *PSynsets*, removing duplicate synsets; add the children and parent synsets to the *PSynsets* list, in this specific order; get all the objects indexed by *Synset* that have the same type (package, class or interface); intersect these objects (*Objects*) with the initial list of objects (*ObjectList*[2]); and finally add the remaining objects to the list of found objects. In the last step, the algorithm ranks *ObjsFound* using the similarity metrics and returns the N best objects.

Basically, what the algorithm does is to start with the *QObj* synset, and then searches for objects indexed by the same synset. If there are not enough objects, the algorithm uses *is-a* relations of this synset to look for objects, in a spreading activation way. When it has found enough objects, it stops and ranks them using the similarity metrics.

Object retrieval has two distinct phases. First the WordNet *is-a* relations are used as an index structure to find relevant objects. Then a similarity metric is used to select the best N objects. This process comprises a trade-off between the first phase which is inexpensive from the computational perspective, and a second phase which is more demanding for computational resources but also more accurate. In the next subsection we present the similarity metrics used for ranking retrieved objects.

The algorithm described before was the first version that was developed. Since

[2] *ObjectList* is a input parameter to the algorithm and it's initial value is chosen by the designer.

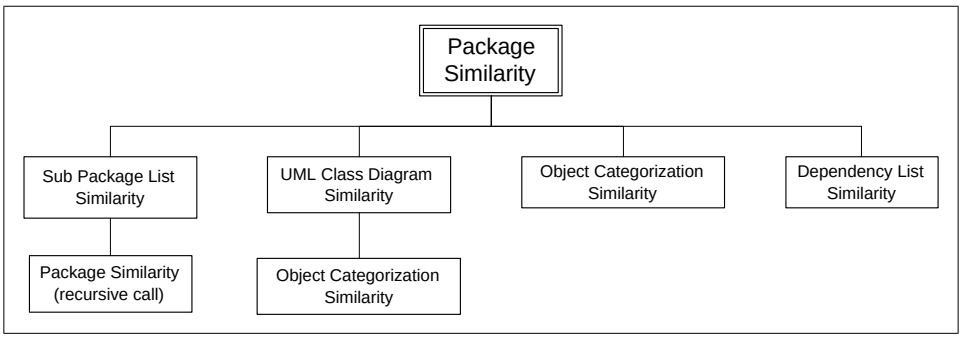


Figure 4.2: An hierarchical representation of the package similarity metrics used in retrieval.

then we have implemented two new retrieval algorithms: an extended version and an ambiguous version. Both work as the previous version (which we call simple version) with the difference that the list of starting synsets (the *PSynsets* list) is different in these new versions. In the simple version the list starts with the synset of the target object. The extended version uses the synset of the target object and all the synsets of the objects that integrate the target problem. This version only works for package retrieval, and in this way the target problem has to be a package. The idea of this version is that similar packages have similar objects. The ambiguous version uses all the target object synsets (there is no disambiguation of the object's name) as the starting synset list. This version bypasses the errors of WSD, but it can also retrieve useless objects if there are 'polysemous' name objects (objects with the same names, but referring to different concepts). Our idea is that if this happens, the similarity metrics will resolve this problem by correctly ranking the objects.

4.1.2 Similarity Metrics

There are three similarity metrics that can be used directly by the retrieval mechanism. These metrics concern to *packages*, *classes* and *interfaces*. Since these metrics are complex and call other similarity metrics, figures 4.2, 4.3 and 4.4 present the hierarchical call scheme for these three metrics.

Package Similarity

The package similarity metric takes several aspects of the UML class diagram into account. The similarity between packages Pk_1 and Pk_2 is defined as:

$$S(Pk_1, Pk_2) = \omega_1 \cdot S(SPs_1, SPs_2) + \omega_2 \cdot S(OBs_1, OBs_2) + \omega_3 \cdot S(S_1, S_2) + \omega_4 \cdot S(D_1, D_2) \quad (4.1)$$

This metric comprises four terms: sub-package list similarity $S(SPs_1, SPs_2)$; UML class diagram similarity $S(OBs_1, OBs_2)$; object categorization similarity[3] $S(S_1, S_2)$; and dependency list similarity $S(D_1, D_2)$. These four items are combined in a weighted sum. Different weight values have been tested (see subsection 5.2.4). We have chosen as default values, 0.07, 0.2, 0.7, 0.03, respectively. From the weight values it is evident that the most important factors for package similarity are the class diagram similarity and the categorization similarity.

Sub-Package List Similarity

The similarity between sub-package lists SPs_1 and SPs_2 is defined as:

$$S(SPs_1, SPs_2) = \frac{\sum_{i=1}^{n} S(SPs_{1i}, SPs_{2i})}{(\#SPs_1 + \#SPs_2)} - \frac{UM(SPs_1) + UM(SPs_2)}{2 \cdot (\#SPs_1 + \#SPs_2)} + \frac{1}{2} \quad (4.2)$$

where n is the number of matching sub-packages, $S(SPs_{1i}, SPs_{2i})$ is the similarity between packages, $UM(SPs_i)$ is the number of unmatched packages in SPs_i, SPs_{ij} is the j element of SPs_i, and $\#SPs_i$ is the number of packages in SPs_i. Special situations are resolved using table 4.1. There are two main factors in this formula: the similarity between matching sub-packages, and the number of unmatched sub-packages. The third part of the formula (the 1/2 factor) is used so that the formula output can range from 0 to 1.

UML Class Diagram Similarity

The similarity between lists of UML objects OBs_1 and OBs_2 is:

$$S(OBs_1, OBs_2) = \frac{\sum_{i=1}^{n} S(OB_{1i}, OB_{2i})}{(\#OBs_1 + \#OBs_2)} - \frac{UM(OBs_1) + UM(OBs_2)}{2 \cdot (\#OBs_1 + \#OBs_2)} + \frac{1}{2} \quad (4.3)$$

[3] Also called type similarity.

Table 4.1: Table used for solving special situations in formulas 4.2, 4.3, 4.7, 4.10 and 4.12. 'x' - any positive integer, '0' - cardinality zero or zero value, 'np' - not possible, and 'F' - given by the formula.

n	0	0	0	0	x	x	x	x
$\#X_1$	0	0	x	x	0	0	x	x
$\#X_2$	0	x	0	x	0	x	0	x
$S(Y_1, Y_2)$	1	0	0	0	np	np	np	F

where OB_{ij} is the j element of OB_i, and $S(OB_{1i}, OB_{2i})$ is the object categorization similarity. $\#OBs_i$ is the number of objects in OBs_i, $UM(OBs_i)$ is the number of unmapped objects in OBs_i, n is the number of matched objects. Special situations are solved using table 4.1.

Object Categorization Similarity

The object object categorization similarity is computed using the objects' synsets. The similarity between synset S_1 and S_2 is:

$$S(S_1, S_2) = \frac{1}{\ln\left(Min\{\forall Path(S_1, S_2)\} + 1\right) + 1} \tag{4.4}$$

where *Min* is the function returning the smallest path between S_1 and S_2. $Path(S_1, S_2)$ is the WordNet path between synset S_1 and S_2, which returns the number of relations between these synsets. *ln* is the natural logarithm function, which is used for a gradual decay of the similarity values. The path between synsets is computed using only *is-a* links.

Dependency List Similarity

Dependencies are UML relation types expressing a dependence between two packages. For instance, package A depends on package B because a class in A uses a class from B. The similarity between dependency lists D_1 and D_2 is given by:

$$S(D_1, D_2) = \omega_1 \cdot \frac{|\#ID_1 - \#ID_2|}{Max\{\#ID_1, \#ID_2\}} + \omega_2 \cdot \frac{|\#OD_1 - \#OD_2|}{Max\{\#OD_1, \#OD_2\}} \tag{4.5}$$

where ω_1 and ω_2 are constants, and $\sum \omega_i = 1$ (default values are: 0.5; 0.5), ID_i is the set of input dependencies in D_i, and OD_i is the set of output dependencies in D_i.

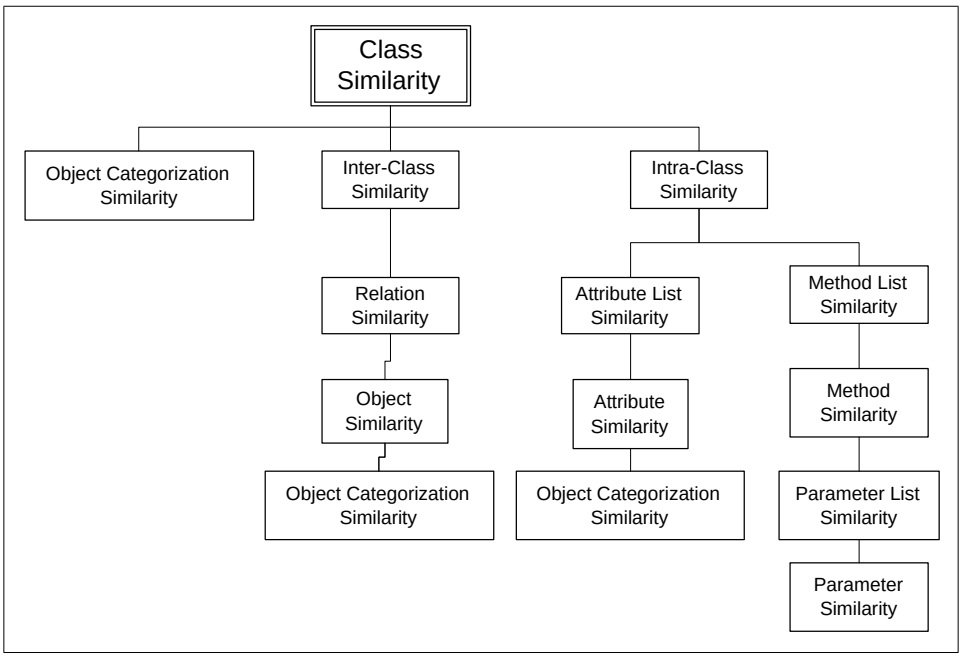


Figure 4.3: An hierarchical representation of the class metrics used in retrieval.

Class Similarity

The class similarity metric is based on WordNet categorization and object structure and comprises three components: categorization similarity, inter-class similarity, and intra-class similarity (see figure 4.3). The similarity between class C_1 and C_2, is defined as:

$$S(C_1, C_2) = \omega_1 \cdot S(S_1, S_2) + \omega_2 \cdot S(Ie_1, Ie_2) + \omega_3 \cdot S(Ia_1, Ia_2) \tag{4.6}$$

where $S(S_1, S_2)$ is the categorization similarity defined as the distance between the synset of C_1 (S_1) and the synset of C_2 (S_2, see equation 4.4). $S(Ie_1, Ie_2)$ is the inter-class similarity based on the similarity between the diagram relations of C_1 and C_2 (see subsection ahead). $S(Ia_1, Ia_2)$ is the intra-class similarity based on the similarity between attributes and methods of C_1 and C_2 (see subsection ahead). Based on experimental results, we use 0.5, 0.25, and 0.25 as the default values for ω_1, ω_2 and ω_3.

Table 4.2: The similarity table for relation cardinality.

	1-N	N-N	1-1
1-N	1	0.3	0.1
N-N	0.3	1	0.3
1-1	0.1	0.3	1

Inter-Class Similarity

The inter-class similarity between two objects (classes or interfaces) is based on matching of the relations in which both objects are involved. The similarity between objects O_1 and O_2 is:

$$S(O_1, O_2) = \frac{\sum_{i=1}^{n} S(R_{1i}, R_{2i})}{(\#R_1 + \#R_2)} - \frac{UM(R_1) + UM(R_2)}{2 \cdot (\#R_1 + \#R_2)} + \frac{1}{2} \tag{4.7}$$

where R_i is the set of relations in object i, R_{ij} is the j element of R_i, n is the number of matched relations, $S(R_{1i}, R_{2i})$ is the relation similarity, which is explained ahead, and $UM(R)$ is the number of unmatched elements in R. Special situations are solved using table 4.1.

Relation Similarity

The similarity between two relations R_1 and R_2 is based on the similarity between source and destination objects, and on the relation properties. The formula for this metric is:

$$S(R_1, R_2) = \omega_1 \cdot S(SO_1, SO_2) + \omega_2 \cdot S(DO_1, DO_2) + \omega_3 \cdot S(Ca_1, Ca_2) + \omega_4 \cdot S(A_1, A_2) \tag{4.8}$$

where $S(SO_1, SO_2)$ is the similarity between source objects, $S(DO_1, DO_2)$ is the similarity between destination objects, both defined ahead, $S(Ca_1, Ca_2)$ is the similarity between the cardinality of the relations, $S(A_1, A_2)$ is the similarity between the aggregations[4] of the relations, and ω_i are constants, with $\sum \omega_i = 1$ (default values are: 0.2; 0.2, 0.5 and 0.1).

The cardinality similarity, $S(Ca_1, Ca_2)$, is given by table 4.2. The aggregation similarity, $S(A_1, A_2)$, is 0 if the relations have different aggregations and 1 otherwise.

[4]An aggregation is a particular instance of an UML association, in which one of the involved objects is part of the other relation object.

Source/Destination Object Similarity

The similarity between source objects or destination objects ($S(O_1, O_2)$) is given by:

$$\begin{cases} S(S_1, S_2) & \Leftarrow T(O_1) = T(O_2)\ , \\ k & \Leftarrow \text{otherwise.} \end{cases}$$

where k is a number between zero and one, which represents the similarity between a class and an interface, we use 0.1 as the default value. $T(O_i)$ returns the object's type: class or interface. $S(S_1, S_2)$ is the object type similarity, see equation 4.4.

Intra-Class Similarity

The intra-class similarity between objects O_1 and O_2 is defined as:

$$S(O_1, O_2) = \omega_1 \cdot S(As_1, As_2) + \omega_2 \cdot S(Ms_1, Ms_2) \tag{4.9}$$

where $S(As_1, As_2)$ is the similarity between attribute lists, $S(Ms_1, Ms_2)$ is the similarity between method lists, both described ahead, ω_1 and ω_2 are constants, with $\sum \omega_i = 1$ (default values are: 0.6; 0.4, based on experimental results).

Attribute List Similarity

The attribute list similarity between lists As_1 and As_2 is based on matching of the attributes of both objects, and is given by:

$$S(As_1, As_2) = \frac{\sum_{i=1}^{n} S(As_{1i}, As_{2i})}{(\#As_1 + \#As_2)} - \frac{UM(As_1) + UM(As_2)}{2 \cdot (\#As_1 + \#As_2)} + \frac{1}{2} \tag{4.10}$$

where As_i is the set of attributes in object i, As_{ij} is the j element of As_i, n is the number of matched attributes, $S(As_{1i}, As_{2i})$ is the attribute similarity, and $UM(As)$ is the number of unmatched attributtes in As. Special situations are solved using table 4.1.

Attribute Similarity

Attribute similarity is based on: the data type, the scope, and the default value. Data type similarity is assessed using the distance of the path between the data types

Table 4.3: Table for computing the scope similarity.

	Public	Package	Protected	Private	Unspecified
Public	1	0.7	0.4	0.1	0
Package	0.7	1	0.7	0.4	0
Protected	0.4	0.7	1	0.7	0
Private	0.1	0.4	0.7	1	0
Unspecified	0	0	0	0	0

being compared. This distance is found using the data type taxonomy. The scope similarity is based on a comparison table, which establishes the similarities between all possible scopes, as defined in UML and Java. The default value similarity is one if the default values are equal or if both do not exist, zero otherwise. The formula is:

$$S(A_1, A_2) = \omega_1 \cdot S(T_1, T_2) + \omega_2 \cdot S(Sc_1, Sc_2) + \omega_3 \cdot S(DV_1, DV_2) \tag{4.11}$$

where ω_i are constants, and $\sum \omega_i = 1$ (default values are: 0.85; 0.1; 0.05), $S(T_1, T_2)$ is the attribute data type similarity given by formula 4.4[5], $S(Sc_1, Sc_2)$ is the scope similarity and is given by table 4.3, and $S(DV_1, DV_2)$ is the default value similarity as described before.

Method List Similarity

The similarity between method lists Ms_1 and Ms_2 is based on the matching of the methods on both lists, and is given by:

$$S(Ms_1, Ms_2) = \frac{\sum_{i=1}^{n} S(Ms_{1i}, Ms_{2i})}{(\#Ms_1 + \#Ms_2)} - \frac{UM(Ms_1) + UM(Ms_2)}{2 \cdot (\#Ms_1 + \#Ms_2)} + \frac{1}{2} \tag{4.12}$$

where Ms_i is the list of methods in object i, Ms_{ij} is the j method of Ms_i, n is the number of matched methods, $S(Ms_{1i}, Ms_{2i})$ is the method similarity, and $UM(Ms)$ is the number of unmatched methods in Ms. Special situations are solved using table 4.1.

Method Similarity

Method similarity is computed based on the scope, the input parameters and the return parameter. The formula is:

$$S(M_1, M_2) = \omega_1 \cdot S(Sc_1, Sc_2) + \omega_2 \cdot S(I_1, I_2) + \omega_3 \cdot S(O_1, O_2) \tag{4.13}$$

[5]The formula is the same, instead of synsets and WordNet, the data type taxonomy is used.

Table 4.4: Table for solving special situations of the parameter list similarity. 'x' - any positive integer, '0' - cardinality zero or zero value, 'np' - not possible, and 'F' - given by the formula.

n	0	0	0	0	x	x	x	x
$\#Ps_1$	0	0	x	x	0	0	x	x
$\#Ps_2$	0	x	0	x	0	x	0	x
$S(Ps_1, Ps_2)$	1	0	0	0	np	np	np	F

where ω_i are constants, and $\sum \omega_i = 1$ (default values are: 0.1; 0.4; 0.5), $S(Sc_1, Sc_2)$ is the scope similarity and is given by table 4.3, $S(I_1, I_2)$ is the similarity of input parameters and $S(O_1, O_2)$ is the similarity of output parameters. Both are given by the parameter list similarity described ahead.

Parameter List Similarity

The similarity between parameter lists Ps_1 and Ps_2 is based on matching of the parameters of both lists, and is given by:

$$S(Ps_1, Ps_2) = \frac{\sum_{i=1}^{n} S(Ps_{1i}, Ps_{2i})}{(\#Ps_1 + \#Ps_2)} - \frac{UM(Ps_1) + UM(Ps_2)}{2 \cdot (\#Ps_1 + \#Ps_2)} + \frac{1}{2} \tag{4.14}$$

where Ps_i is the set of parameters in list i, Ps_{ij} is the j parameter of Ps_i, n is the number of matched parameters, $S(Ps_{1i}, Ps_{2i})$ is the parameter similarity, described ahead, and $UM(Ps)$ is the number of unmatched elements in Ps. Special situations are solved using table 4.4.

Parameter Similarity

Parameter similarity is computed based on data types of parameters. The formula for parameter similarity is:

$$S(S_1, S_2) = \frac{1}{\ln\left(Min\{\forall Path(DT_1, DT_2)\} + 1\right) + 1} \tag{4.15}$$

where *Min* is the function returning the smallest path between data types DT_1 and DT_2. $Path(DT_1, DT_2)$ is the path between data type DT_1 and DT_2 in the data type taxonomy, which returns the number of relations between data types. *ln* is the natural logarithm function.

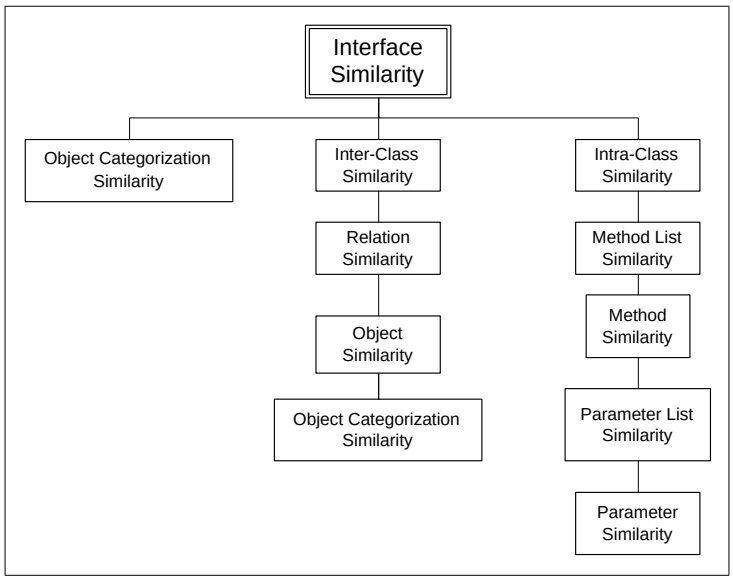


Figure 4.4: An hierarchical representation of the interface metrics used in retrieval.

Interface Similarity

The interface similarity metric is similar to the similarity class metric with the difference that the intra-class similarity metric is only based on the method similarity, since interfaces do not have attributes (see figure 4.4).

4.1.3 Matching Algorithms

This subsection presents the algorithms used in REBUILDER to match lists of: packages, objects, relations, attributes, methods, and parameters. These algorithms are used by the similarity metrics dealing with lists of elements. For instance, the sub package list similarity metric (see subsection 4.1.2) receives two lists of packages and has to establish a mapping between the elements of these lists. The next subsections describe the matching algorithms.

Package Matching Algorithm

The package matching algorithm is used in the sub package list similarity metric (see subsection 4.1.2) to match two lists of packages. The input are two lists: LP_1 and LP_2, it returns: a list of package mappings, the respective mapping similarities, and

1. $Mappings \leftarrow \varnothing$
2. FORALL Package (Pk_1) in LP_1 DO
3. $List_1 \leftarrow$ Get all packages with equivalent classification as Pk_1 in LP_2
4. $Ranked_1 \leftarrow$ Rank $List_1$ by similarity to Pk_1
5. IF $Ranked_1 \neq \emptyset$ THEN
6. $Pk_2 \leftarrow$ Best package in $Ranked_1$
7. Add $(Pk_1, Pk_2, Sim(Pk_1, Pk_2))$ to $Mappings$
8. Remove Pk_1 from LP_1 and Pk_2 from LP_2
9. ENDIF
10. ENFOR
11. RETURN $Mappings$, $\#LP_1$

Figure 4.5: The package matching algorithm used in REBUILDER.

the number of unmatched packages in LP_1. The algorithm is presented in figure 4.5. Note that two objects have equivalent classification if they have the same name and synsets.

The algorithm starts by initializing the mappings list ($Mappings$) as empty, and then goes into a cycle, where for each package (Pk_1) in LP_1 it performs the following operations: get all the packages in LP_2 with the same classification as Pk_1, that is, packages with the same synset; rank the found packages by similarity with Pk_1; if the resulting list is not empty, then the best package is chosen, mapped to Pk_1, and the mappings added to $Mappings$; the mapped packages are removed from the respective lists. At the end of the cycle the $Mappings$ and the number of unmapped packages in LP_1 are returned.

Package Objects Matching Algorithm

During the assessment of the diagram similarity (see subsection 4.1.2) it is necessary to match the UML objects from two lists. Object matching uses two algorithms, both based on structural matching. One is guided by relation mapping (see figure 4.6), and the other is guided by object matching (see figure 4.7). These algorithms are the core of the mapping phase for analogical reasoning (see section 4.2).

Considering a problem (or query) diagram that has to be matched with a case diagram, the relation guided algorithm starts by choosing the best relation from the problem diagram based on an independence measure, which is described in the next section. This relation is added to the current relation list (*Relations*). Then the list of mappings (*MappingList*) is initialized as empty. The algorithm then enters a cycle while there are relations in *Relations* to be processed, comprising: get the best relation from *Relations* based on the independence measure; select the best

1. *Relations* ← Get best relation from the query diagram based on independence measure
2. *MappingList* ← ∅
3. WHILE *Relations* ≠ ∅ DO
4. *PRelation* ← Get best relation from *Relations* based on the independence measure
5. *CRelation* ← Get best matching relation from the base case relations that are candidates (structural constraints must be met)
6. *Mapping* ← Get the mapping between objects, based on the mapping between *PRelation* and *CRelation*
7. Add *Mapping* to *MappingList*
8. Remove *PRelation* from *Relations*
9. Add to *Relations* all the same type relations adjacent to *PRelations*, which are not already mapped (if *PRelation* connects *A* and *B* then the adjacent relations of *PRelations* are the relations in which *A* or *B* are part of, excluding *PRelation*)
10. ENDWHILE
11. Return *MappingList*

Figure 4.6: The object matching algorithm guided by relation mapping.

1. *Objects* ←Get object from query diagram which has the highest independence measure
2. *MappingList* ← ∅
3. WHILE *Objects* ≠ ∅ DO
4. *PObject* ← Get best object from *Objects*, based on the object's independence measure
5. *CObject* ← Get best matching object from the base case objects that are matching candidates (structural constraints must be met)
6. *Mapping* ← Get the mapping between *PObject* and *CObject*
7. Remove *PObject* from *Objects*
8. Remove *CObject* from the base case objects that are matching candidates.
9. Add *Mapping* to *MappingList*
10. Add to *Objects* all the objects adjacent to *PObject* which are not already mapped (an adjacent object *B* to an object *A*, is every object that has a relation with *A*).
11. ENDWHILE
12. Return *MappingList*

Figure 4.7: The matching algorithm guided by object mapping.

matching relation from the case relations, taking into account structural constraints; the objects in both relations are mapped and they are added to the *MappingList*; the mapped relations are removed from the respective lists; the adjacent relations are added to the *Relations* list expanding the mapping. At the end the algorithm returns the *MappingList*.

The second matching algorithm, guided by object mappings, takes as input a problem list comprising UML objects and a case[6] object list. The algorithm starts by selecting the best object from the problem list and adds it to the list of current objects (*Objects*). The best object is the one which has the highest independence measure. The list of mappings (*MappingList*) is setup to empty. Then the algorithm enters into a cycle while there are objects to be processed in *Objects*. The instructions performed in this cycle are: get the best object from *Objects* based on the independence measure; choose the best matching object from the case object list, based on the structural constraints and on the semantic distance between both objects' synsets; map the

[6]Also called base case, since it is the case used for matching.

chosen objects and add this mapping to the $MappingList$; remove both objects from the respective lists; add the adjacent objects that are not mapped to the query object to $Objects$. At the end the algorithm returns the $MappingList$.

Relation Matching Algorithm

This algorithm is used in the inter-class similarity metric (see subsection 4.1.2) to match the problem relations with the retrieved case relations. The input are two lists: LR_1 and LR_2. It returns a list of relation mappings with the respective mapping similarities and the number of unmatched relations in LR_1 discriminated by the type of relation. The algorithm is presented in figure 4.8.

The relation matching algorithm performs the same process for each of the four types of relations: associations, generalizations, dependencies and realizations. The list of mappings ($Mappings$) is setup to empty. Then the algorithm gets all associations, generalizations, dependencies and realizations in LR_1, respectively, to lists A_1, G_1, D_1 and R_1, and all the associations, generalizations, dependencies and realizations in LR_2, respectively, to lists A_2, G_2, D_2 and R_2. The relations in A_1 are ranked using the independence measure. Then for each relation (Rel_1) in A_1 it is performed the following: rank the relations in A_2 regarding to the similarity with Rel_1; if this results into relations in the ranked list, then select the best one (Rel_2) and map it to Rel_1 and add it to $Mappings$; both relations are removed from the respective lists. Then the same mapping process is performed for generalizations (lines 19 to 27), dependencies (lines 28 to 36) and realizations (lines 37 to 45). At the end the algorithm returns $Mappings$ and the number of unmatched relations discriminated by relation type.

Attribute Matching Algorithm

The attribute matching algorithm is used in the attribute list similarity metric (see subsection 4.1.2) to match two lists of attributes. The input comprises two lists: LA_1 and LA_2. It returns a list of attribute mappings with the respective mapping similarities and the number of unmatched attributes in LA_1. The algorithm is presented in figure 4.9.

1. $Mappings \leftarrow \varnothing$
2. $A_1 \leftarrow$ Get all associations from LR_1
3. $G_1 \leftarrow$ Get all generalizations from LR_1
4. $D_1 \leftarrow$ Get all dependencies from LR_1
5. $R_1 \leftarrow$ Get all realizations from LR_1
6. $A_2 \leftarrow$ Get all associations from LR_2
7. $G_2 \leftarrow$ Get all generalizations from LR_2
8. $D_2 \leftarrow$ Get all dependencies from LR_2
9. $R_2 \leftarrow$ Get all realizations from LR_2
10. Rank A_1 using the independence measures
11. FORALL Relation (Rel_1) in A_1 DO
12. $Ranked_1 \leftarrow$ Rank A_2 in relation to Rel_1
13. IF $Ranked_1 \neq \emptyset$ THEN
14. $Rel_2 \leftarrow$ Best relation in $Ranked_1$
15. Add $(Rel_1, Rel_2, Sim(Rel_1, Rel_2))$ to $Mappings$
16. Remove Rel_1 from A_1 and Rel_2 from A_2
17. ENDIF
18. ENFOR
19. Rank G_1 using the independence measures
20. FORALL Relation (Rel_1) in G_1 DO
21. $Ranked_1 \leftarrow$ Rank G_2 in relation to Rel_1
22. IF $Ranked_1 \neq \emptyset$ THEN
23. $Rel_2 \leftarrow$ Best relation in $Ranked_1$
24. Add $(Rel_1, Rel_2, Sim(Rel_1, Rel_2))$ to $Mappings$
25. Remove Rel_1 from G_1 and Rel_2 from G_2
26. ENDIF
27. ENFOR
28. Rank D_1 using the independence measures
29. FORALL Relation (Rel_1) in D_1 DO
30. $Ranked_1 \leftarrow$ Rank D_2 in relation to Rel_1
31. IF $Ranked_1 \neq \emptyset$ THEN
32. $Rel_2 \leftarrow$ Best relation in $Ranked_1$
33. Add $(Rel_1, Rel_2, Sim(Rel_1, Rel_2))$ to $Mappings$
34. Remove Rel_1 from D_1 and Rel_2 from D_2
35. ENDIF
36. ENFOR
37. Rank R_1 using the independence measures
38. FORALL Relation (Rel_1) in R_1 DO
39. $Ranked_1 \leftarrow$ Rank R_2 in relation to Rel_1
40. IF $Ranked_1 \neq \emptyset$ THEN
41. $Rel_2 \leftarrow$ Best relation in $Ranked_1$
42. Add $(Rel_1, Rel_2, Sim(Rel_1, Rel_2))$ to $Mappings$
43. Remove Rel_1 from R_1 and Rel_2 from R_2
44. ENDIF
45. ENFOR
46. RETURN $Mappings$, $\#A_1 + \#G_1 + \#D_1 + \#R_1$

Figure 4.8: The relation matching algorithm used in REBUILDER.

1. $Mappings \leftarrow \varnothing$
2. FORALL Attribute (A_1) in LA_1 DO
3. $List_1 \leftarrow$ Get all attributes from LA_2 with the same synset as A_1
4. $Ranked_1 \leftarrow$ Rank $List_1$ by similarity to A_1
5. IF $Ranked_1 \neq \emptyset$ THEN
6. $A_2 \leftarrow$ Best attribute in $Ranked_1$
7. Add $(A_1, A_2, Sim(A_1, A_2))$ to $Mappings$
9. Remove A_1 from LA_1 and A_2 from LA_2
10. ENDIF
11. ENFOR
12. RETURN $Mappings$, $\#LA_1$

Figure 4.9: The attribute matching algorithm used in REBUILDER.

```
1. Mappings ← ∅
2. FORALL Method (M1) in LM1 DO
3.       List1 ← Get all methods from LM2 with the same Input and Output parameters as M1
4.       Ranked1 ← Rank List1 by similarity to M1
5.       IF Ranked1 ≠ ∅ THEN
6.             M2 ← Best method in Ranked1
7.             Add (M1, M2, Sim(M1, M2)) to Mappings
8.             Remove M1 from LM1 and M2 from LM2
9.       ENDIF
10. ENFOR
11. RETURN Mappings, #LM1
```

Figure 4.10: The method matching algorithm used in REBUILDER.

The attribute matching algorithm starts by initializing the list of mappings ($Mappins$) to empty. Then for each attribute (A_1) in LA_1 it performs: get all attributes in LA_2 with the same synset as A_1; rank the resulting list based on the similarity with A_1 (note that the name is also taken into account); if this results into attributes in the ranked list, then select the best one (A_2) and map it with A_1, adding it to $Mappings$; remove both attributes from the respective lists. At the end the algorithm returns $Mappings$ and the number of elements in LA_1.

Method Matching Algorithm

The method matching algorithm is used in the method list similarity metric (see subsection 4.1.2) to match two lists of methods. The input are two lists: LM_1 and LM_2. It returns a list of method mappings with the respective mapping similarities and the number of unmatched methods in LM_1. The algorithm is presented in figure 4.10.

The method matching algorithm starts by initializing the list of mappings ($Mappins$) to empty. Then for each method (M_1) in LM_1 it performs: get all the methods in LM_2 with the same input and output parameters as M_1; rank the resulting list based on the similarity with M_1; if this results into methods in the ranked list, then select the best one (M_2) and map it with M_1, adding the mapping to $Mappings$; both methods are removed from the respective lists. At the end the algorithm returns $Mappings$ and the number of elements in LM_1.

1. $Mappings \leftarrow \varnothing$
2. FORALL Parameter (P_1) in LP_1 DO
3. $List_1 \leftarrow$ Get all parameters in LP_2 with the same name as P_1
4. $Ranked_1 \leftarrow$ Rank $List_1$ by similarity to P_1
5. IF $Ranked_1 \neq \emptyset$ THEN
6. $P_2 \leftarrow$ Best parameter in $Ranked_1$
7. Add $(P_1, P_2, Sim(P_1, P_2))$ to $Mappings$
8. Remove P_1 from LP_1 and P_2 from LP_2
9. ENDIF
10. ENFOR
11. RETURN $Mappings$, $\#LP_1$

Figure 4.11: The parameter matching algorithm used in REBUILDER.

Parameter Matching Algorithm

The parameter matching algorithm is used in the parameter list similarity metric (see subsection 4.1.2) to match two lists of parameters. The input comprises two lists: LP_1 and LP_2, it returns a list of parameter mappings with the respective mapping similarities and the number of unmatched parameters in LP_1. The algorithm is presented in figure 4.11. It is similar to the method matching algorithm.

4.1.4 Independence Measures

This section describes the independence measures used in REBUILDER to reflect the independence between objects in the class diagram. There are three independence measures concerning: classes, interfaces and relations. The independence of classes and interfaces are defined in the same way.

Class/Interface Independence

The independence of a class C_1 (similarly for an interface) is defined by:

$$Ind(C_1) = \frac{C(C_1, As) + C(C_1, Gs) + C(C_1, Ds) + C(C_1, Rs)}{\#RDiagram} \tag{4.16}$$

where $\#RDiagram$ is the total number of relations in the diagram, and $C(C_1, As)$ is the independence given to class C_1 due to it's associations (As) and is computed by:

$$C(C_1, As) = \sum_{i=1}^{\#As} C(C_1, As_i) \tag{4.17}$$

where As_i is the *ith* association relation of As and $C(C_1, As_i)$ is defined by:

$$\begin{cases} 1 & \Leftarrow As_i \text{ is an association, with cardinality N-N or 1-N,} \\ 0 & \Leftarrow As_i \text{ is an association, with cardinality N-1 or 1-1.} \end{cases}$$

The idea is that a class C_1 with a cardinality N on the opposite side to it, is more general (and thus more independent) than if it has cardinality 1 on the opposite side.

$C(C_1, Gs)$ is the independence given to class C_1 due to it's generalizations (Gs), and is computed by:

$$C(C_1, Gs) = \sum_{i=1}^{\#Gs} C(C_1, Gs_i) \tag{4.18}$$

where Gs_i is the ith generalization relation of Gs, and $C(C_1, Gs_i)$ is defined by:

$$\begin{cases} 1 & \Leftarrow \text{otherwise,} \\ 0 & \Leftarrow C_1 \text{ is subclass in } Gs_i. \end{cases}$$

The idea is that in a generalization relation, the superclass is more independent than the subclass.

$C(C_1, Ds)$ is the independence given to class C_1 due to it's dependencies (Ds), and is computed by:

$$C(C_1, Ds) = \sum_{i=1}^{\#Ds} C(C_1, Ds_i) \tag{4.19}$$

where Ds_i is the ith dependency relation of Ds, and $C(C_1, Ds_i)$ is defined by:

$$\begin{cases} 1 & \Leftarrow C_1 \text{ is not the dependable class in } Ds_i, \\ 0 & \Leftarrow C_1 \text{ is the dependable class in } Ds_i. \end{cases}$$

The idea in this type of relation is evident. The dependent class is the less independent, thus having a lower score.

Dependency between classes may not be explicitly stated by an UML dependency relation. To infer these implicit dependencies we use the class methods and attributes. Class X is the independent class and class Y is the dependent one, if:

- Y has an attribute such as that it's data type is X, or
- Y has a method or methods with input or output parameter whose data type is X.

$C(C_1, Rs)$ is the independence given to class C_1 due to it's realizations (Rs), and is defined as:

$$C(C_1, Rs) = \sum_{i=1}^{\#Rs} C(C_1, Rs_i) \tag{4.20}$$

where Rs_i is the ith realization relation of Rs, and $C(C_1, Rs_i)$ is defined as:

$$\begin{cases} 1 & \Leftarrow C_1 \text{ is not the interface in } Rs_i, \\ 0 & \Leftarrow C_1 \text{ is the interface in } Rs_i. \end{cases}$$

The idea is that in a realization, the interface is always dependant on the implementation class, which makes it less independent.

The independent measure has the goal of determining the degree of dependency that a class has in relation to the other diagram objects. This is useful in the sense that it can be used to identify which are the more important objects in the diagram. The matching algorithms use it to rank the lists of objects, so that the most independent objects of one list, match the equivalent ones in the other list, provided this correspondence exists.

In the specific case of class independency what is examined is the type of relations that a class has with the other diagram objects, trying to identify the dependence direction of each relation. In the case of generalizations, realizations or dependencies, this is clear since the relation represents a sort of dependence between objects. In the case of an association this is more difficult to establish. We have come up with a possible decision based on the association's cardinality, but this is an open issue.

Relation Independence

The independence of a relation R_1 is defined by:

$$Ind(R_1) = \frac{Ind(SO) + Ind(DO)}{2} \tag{4.21}$$

where SO is the source object of the relation and DO is the destination object.

4.1.5 A Retrieval Example

In this section we present an example of case retrieval using WordNet. The example consists on using an UML diagram (*P1* from figure 4.12) as the problem, and then using the package retrieval algorithm to retrieve relevant cases from the case library. We use an hypothetical case library comprising six cases (*C1* to *C6*, see figure 4.12), indexed according to figure 4.13. Each node in the tree represents a synset, the word (or words) represent the intended meaning and the numbers in square brackets are the synset IDs, which univocally represent the synsets in WordNet. The squared

nodes represent case indexing. Case *C1* and *C2* are indexed under the university synset, while case *C3* is indexed under the school synset. For simplification reasons, we omitted cases' attributes and methods.

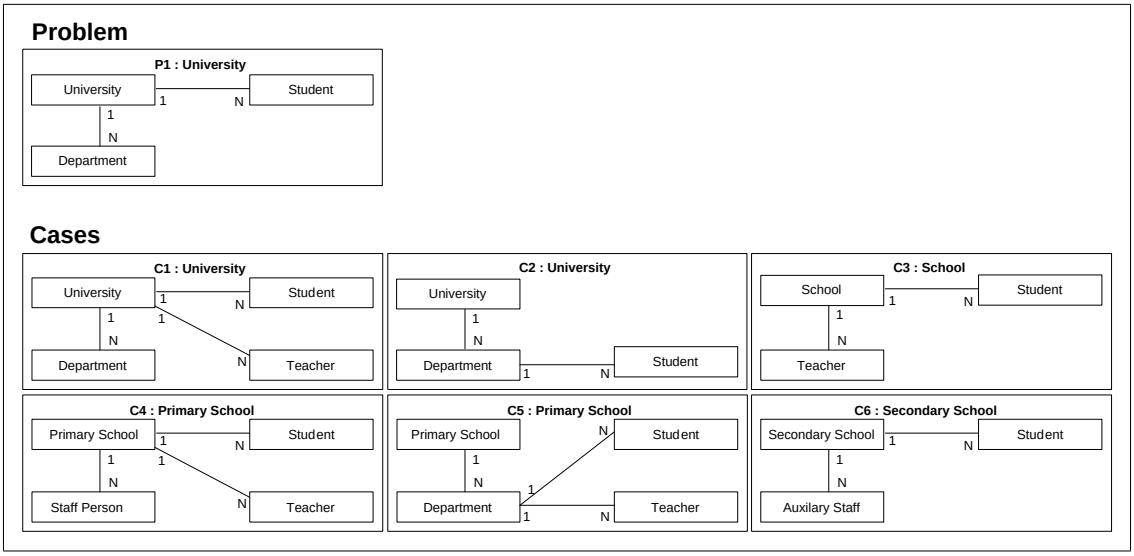


Figure 4.12: The UML class diagrams used in the retrieval example (attributes and methods were omitted).

Supposing that the number of cases to be found is 3, the cases that will be retrieved are *C1*, *C2* and *C3*. Table 4.5 describes the retrieval algorithm trace, showing the contents of four main variables: *Synset* - the synset currently being explored; $ObjsExplored$ - the objects that were explored, this includes the current synset; $ObjsFound$ - objects found by the algorithm and that are going to be retrieved; $PSynsets$ - the list of synsets to be explored.

After the retrieval, cases must be ranked by package similarity. For each case a similarity measure between *P1* and each case is calculated. The established mappings are:

- Between *P1* and *C1*: *University-University* with score 1; *Department-Department* with score 1; *Student-Student* with score 1.
- Between *P1* and *C2*: *University-University* with score 1; *Department-Department* with score 1.
- Between *P1* and *C3*: *University-School* with score 0.69; *Department-Teacher* with score 0.4; *Student-Student* with score 1.

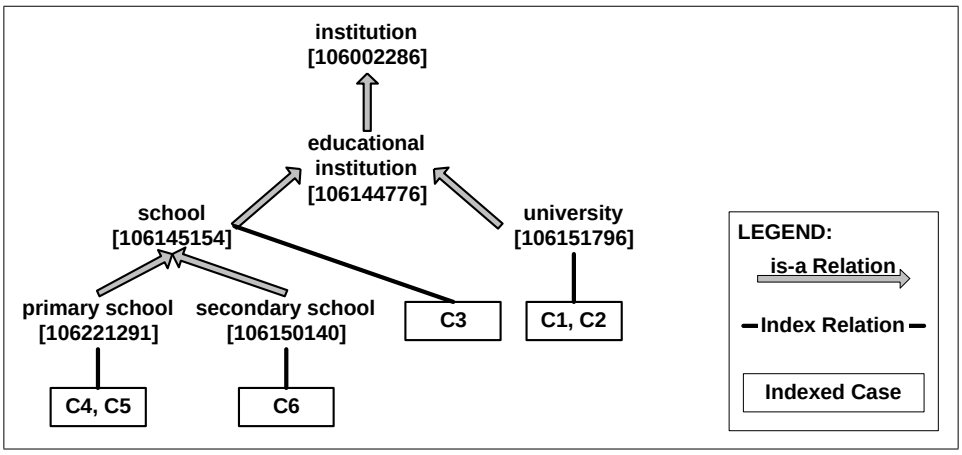


Figure 4.13: Extract of WordNet *is-a* hierarchy and case indexing.

Table 4.5: Iterations made by the retrieval algorithm, for the retrieval example.

Iteration	***Synset***	***ObjsExplored***	***ObjsFound***	***PSynsets***
0	$\varnothing$	$\varnothing$	$\varnothing$	$\{university\}$
1	$university$	$\{university\}$	$\{C_1, C_2\}$	$\{educational\ institution\}$
2	$educational\ institution$	$\{university$ $, educational\ institution\}$	$\{C_1, C_2\}$	$\{school$ $, institution\}$
3	$school$	$\{university, educational$ $institution, school\}$	$\{C_1, C_2, C_3\}$	$\{institution, primary$ $school, secondary\ school\}$

Next we present the computations for the similarity measures involved in the ranking of cases *C1*, *C2* and *C3*.

$$\begin{aligned}
\mathbf{S(P1,C1)} &= 0.07 \cdot 0 + 0.4 \cdot S(OBs_{P1}, OBs_{C1}) + 0.5 \cdot S(T_{P1}, T_{C1}) + 0.03 \cdot 0 = \mathbf{0.84} \\
\mathbf{S}(OBs_{P1}, OBs_{C1}) &= \frac{1+1+1}{3+4} - \frac{0+1}{2\cdot(3+4)} + \frac{1}{2} = 0.86 \\
\mathbf{S}(T_{P1}, T_{C1}) &= S(university, university) = \frac{1}{\ln(0+1)+1} = 1
\end{aligned} \tag{4.22}$$

$$\begin{aligned}
\mathbf{S(P1,C2)} &= 0.07 \cdot 0 + 0.4 \cdot S(OBs_{P1}, OBs_{C2}) + 0.5 \cdot S(T_{P1}, T_{C2}) + 0.03 \cdot 0 = \mathbf{0.77} \\
\mathbf{S}(OBs_{P1}, OBs_{C2}) &= \frac{1+1}{3+3} - \frac{1+1}{2\cdot(3+3)} + \frac{1}{2} = 0.67 \\
\mathbf{S}(T_{P1}, T_{C2}) &= S(university, university) = \frac{1}{\ln(0+1)+1} = 1
\end{aligned} \tag{4.23}$$

$$\begin{aligned}
\mathbf{S(P1,C3)} &= 0.07 \cdot 0 + 0.4 \cdot S(OBs_{P1}, OBs_{C3}) + 0.5 \cdot S(T_{P1}, T_{C3}) + 0.03 \cdot 0 = \mathbf{0.58} \\
\mathbf{S}(OBs_{P1}, OBs_{C3}) &= \frac{0.69+0.4+1}{3+3} - \frac{0+0}{2\cdot(3+3)} + \frac{1}{2} = 0.85 \\
\mathbf{S}(T_{P1}, T_{C3}) &= S(university, school) = \frac{1}{\ln(2+1)+1} = 0.48
\end{aligned} \tag{4.24}$$

Note that in the package similarity the sub package similarity is zero because there are no sub packages, the same happens with the dependency similarity. The second and third equations in each formula, respectively, present the class diagram similarity, and the package type similarity. The ranking is: *C1* with a similarity 0.84, *C2* with 0.77, and *C3* with 0.58. From the formulas we can also see that *C3* has a better class diagram similarity than *C2* due to better object matching. Though in the package similarity this is cancelled by the package type similarity. Weights can be adjusted to select the most important factor. In chapter 5 we explore the space of possible weights, looking for a good configuration.

4.2 Analogy Module

Analogical reasoning is used in REBUILDER to suggest class diagrams to the designer, based on a query diagram. The analogy process comprises three steps:

1. Identify candidate diagrams for analogy.
2. Map the candidate diagrams.
3. Create new diagrams, by knowledge transfer between a candidate diagram and the query.

These steps will be detailed in the next subsections.

4.2.1 Candidate Selection

Cases are selected from the case library to be used as source diagrams. The selected candidates must be appropriate, that is, they should have structural and semantic similarity with the target diagram, otherwise the whole mapping phase can be at risk. Six alternative strategies for candidate selection can be used in REBUILDER (see table 4.6).

Candidate selection is decomposed in two subtasks: retrieval and ranking. While retrieval should be a fast comparison task yielding a subset of candidate cases, ranking

is a more complex and computationally demanding phase where cases are ordered accordingly to a defined criterium.

Candidate retrieval must be a computationally inexpensive task, with the main purpose of identifying a first set of cases that are possible candidates for analogy mapping with the target problem. Analogy uses the retrieval algorithm described in section 4.1.1 to select the analog candidates. This algorithm returns N cases that are then ranked.

Candidate ranking is based on metrics that incorporate several criteria assessing semantic and structural similarity. Three different similarity metrics are used: one that combines semantic and structure evaluation, a second one that uses an independence measure, and a third one that evaluates structural properties.

The first similarity metric is the package similarity metric, used in case retrieval (see section 4.1.2). This metric compares two UML class diagrams based on two items: diagram synset (the package's synset, which is a semantic evaluation) and diagram structure (combines structural evaluation of the diagram with object synset distances, it is both structural and semantical).

The independence measure is specific to UML and evaluates the independence of each object in the class diagram based on the structural relations of diagram objects (see subsection 4.1.4). Each class or interface in the class diagram is assigned a score that reflects the independence level of that object from other objects in the diagram, thus yielding a sense of importance of that object in the diagram.

The third similarity metric is based on diagram structural properties, which is based on the work of O'Donoghue [O'Donoghue and Crean, 2002]. The main idea of this metric is to use the structural properties of two diagrams to determine their structural similarity. O'Donoghue argues that structural similarity is more important for the identification of good analogical candidates than semantic similarity. But because the best structural similarity can only be assessed using structural mapping, which is computational expensive, he uses structural properties to assess the structural similarity, which are fast to compute [Crean and O'Donoghue, 2002].

The structural properties that we have used are:

- Number of loops in the diagram.

- Average loop size.
- Number of classes.
- Number of interfaces.
- Number of generalizations.
- Number of realizations.
- Number of associations.
- Number of dependencies.
- Independence metric comparison (described before).

Then we have devised six different candidate selection strategies (see also table 4.6):

Strategy 1: First uses the semantic retrieval to retrieve a set of cases, which are then ranked by the first similarity metric (package similarity metric). This is the first implemented strategy in REBUILDER, but it is also the most complex.

Strategy 2: Uses the semantic retrieval and then uses the independence measure to rank the retrieved cases.

Strategy 3: Uses the semantic retrieval, but then, retrieved cases are ranked using the third similarity metric (based on structural properties).

Strategy 4: Does not use a retrieval algorithm, instead it applies the similarity metric to all cases. In this strategy the first similarity metric is used (this strategy is the most expensive in computational terms, it is used only for comparison with the other strategies).

Strategy 5: Similar to strategy 4. The ranking is performed by the independence measure.

Strategy 6: Similar to strategy 4. Ranks all cases in the case library using the third similarity metric (this strategy is also implemented to be compared with the other ones).

Section 5.3.3 presents experimental work comparing these strategies.

Table 4.6: The retrieval strategies for analogical reasoning implemented in REBUILDER.

Strategy	Retrieval	Ranking
1	Semantic	Semantic and Structural
2	Semantic	Structural (independence measure)
3	Semantic	Structural properties
4	No	Semantic and Structural
5	No	Structural (independence measure)
6	No	Structural properties

4.2.2 Mapping Process

The second phase of analogy is the mapping of each candidate case with the query diagram, yielding an object list mapping for each candidate case. This phase relies on two alternative algorithms: one guided by relation mapping, and the other by object mapping, both return a list of mappings between objects.

Relation-Based Mapping

The relation-based algorithm (see figure 4.14) uses the UML relations to establish the object mappings. In line 1, the algorithm starts the mapping process by selecting a query relation based on the independence measure, which chooses the relation that connects the two most important diagram objects (the ones with the highest independence score). The *MappingList* is initialized to an empty list (line 2). Then it tries to find a matching relation on the candidate diagram, by getting the best relation from the query diagram (line 4), getting the best relation from the base case (line 5), and then mapping the object relations (line 6). After this mapping, it starts the mapping by the neighbor relations, spreading the mapping using the diagram relations (lines 7, 8 and 9). In the end it returns the list of mappings (line 10). This algorithm maps objects in pairs corresponding to the relation's objects.

Object-Based Mapping

The object-based algorithm (see figure 4.15) starts the mapping selecting the most independent query object, based on the UML independence heuristic. After finding the corresponding candidate object, it tries to map the neighbor objects of the query

1. *Relations* ← Get best relation from the query diagram based on independence measure
2. *MappingList* ← ∅
3. WHILE *Relations* ≠ ∅ DO
4. *PRelation* ← Get best relation from *Relations* based on the independence measure
5. *CRelation* ← Get best matching relation from base case relations that are matching candidates (structural constraints must be met)
6. *Mapping* ← Get the mapping between objects, based on the mapping between *PRelation* and *CRelation*
7. Remove *PRelation* from *Relations*
8. Add *Mapping* to *MappingList*
9. Add to *Relations* all the same type relations adjacent to *PRelations*, which are not already mapped (if *PRelation* connects *A* and *B* then the adjacent relations of *PRelations* are the relations in which *A* or *B* are part of, excluding *PRelation*)
10. ENDWHILE
11. Return *MappingList*

Figure 4.14: The relation-based mapping algorithm.

1. *Objects* ←Get object from the query diagram with the highest independence measure
2. *MappingList* ← ∅
3. WHILE *Objects* ≠ ∅ DO
4. *PObject* ← Get best object from *Objects*, based on the object's independence measure
5. *CObject* ← Get best matching object from the base case objects that are matching candidates (structural constraints must be met)
6. *Mapping* ← Get the mapping between *PObject* and *CObject*
7. Remove *PObject* from *Objects*
8. Add *Mapping* to *MappingList*
9. Add to *Objects* all the objects adjacent to *PObject* which are not already mapped (an adjacent object *B* to an object *A*, is every object that has a relation with *A*).
10. ENDWHILE
11. Return *MappingList*

Figure 4.15: The object-based mapping algorithm.

object, taking the object's relations as constraints in the mapping. This algorithm and the one guided by relation mappings are described in detail, in section 4.1.3, where they are used as matching algorithms.

Mapping Ranking Criteria

Both mapping algorithms previously presented, satisfy the structural constraints defined by the UML diagram relations. But most of the resulting mappings do not map all the problem objects, so the mappings can be ranked using four different ranking metrics:

- Based on the number of mapped objects:

$$\frac{K}{PObjs} \tag{4.25}$$

  where K is the number of mapped objects, and $PObjs$ is the number of objects in the problem.

- Based on the independence sum of mapped objects in the problem:

$$\frac{\sum_{i=1}^{m} Ind(MappedPObj_i)}{\sum_{j=1}^{n} Ind(PObj_j)} \tag{4.26}$$

  where $Ind(X)$ represents the independent heuristic of X, $MappedPObj_i$ is the ith mapped objects in the problem, $PObj_j$ is the jth object in the problem, m is the number of mapped objects in the problem, and n is the number of objects in the problem.

- Based on the independence sum of mapped objects in the problem and the case:

$$1 - \sum_{i=1}^{k} \left| \frac{PM_i}{\sum_{j=1}^{k} PM_j} - \frac{CM_i}{\sum_{j=1}^{k} CM_j} \right| \tag{4.27}$$

  Where PM_i is the independence value of mapped problem object i. CM_i is the independence value of mapped case object i.

- Based on the number of mapped objects and independence sum:

$$\omega_1 \cdot \frac{K}{PObjs} + \omega_2 \cdot \left(1 - \sum_{i=1}^{k} \left| \frac{PM_i}{\sum_{j=1}^{k} PM_j} - \frac{CM_i}{\sum_{j=1}^{k} CM_j} \right| \right) \tag{4.28}$$

  We used 0.5 for ω_1 and 0.5 for ω_2, based on experimental work.

The KB Administrator can select the ranking that will be used. Some guidelines are provided in section 5.3.1.

Similarity Metric for Mapping Objects

An important issue in the mapping stage is: which objects to map? Most of the time, there are several candidate objects for mapping with the problem object. To solve this issue, we have developed a metric that is used to choose the mapping candidate. Because we have two mapping algorithms, one based on relations and another on objects, there are two metrics: one for objects and another for relations. These metrics are based on the WordNet distance between the object's synsets, and the relative position of these synsets in relation to the most specific common abstraction concept (see figure 4.16). This subsection describes the metric for ranking objects. The next subsection describes the metric for ranking relations.

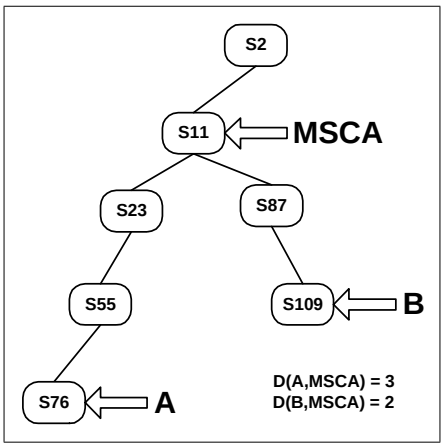


Figure 4.16: An illustration of the MSCA concept.

The similarity metric for mapping two objects (A and B) is based on three factors. The first one is the distance between A's synset and B's synset in the WordNet ontology (D_1). For the second factor, the Most Specific Common Abstraction ($MSCA$) between A and B synsets must be found (see figure 4.16 for an illustration). Considering the distance between A's synset and $MSCA$ ($D(A, MSCA)$), and the distance between B's synset and $MSCA$ ($D(B, MSCA)$), then the second factor is the relation between these two distances (D_2). This factor measures the difference between the level of abstraction of concept A and the level of abstraction of concept B. The last factor is the relative depth of $MSCA$ in the WordNet ontology (D_3), which determines the objects' level of abstraction. The formulas for these measures are described in the following equations.

The similarity between A and B is defined as:

$$\begin{cases} \omega_1 \cdot D_1 + \omega_2 \cdot D_2 + \omega_3 \cdot D_3 & \Leftarrow \text{exists } MSCA \text{ between } A \text{ and } B, \\ 0 & \Leftarrow \text{otherwise.} \end{cases}$$

where ω_1, ω_2 and ω_3 are weights associated with each factor. The values are selected based on empirical work and are: 0.55, 0.3, and 0.15. The factor D_1, which measures the distance between A and B synsets, is:

$$D_1 = 1 - \frac{D(A, B)}{2 \cdot DepthMax} \tag{4.29}$$

where $DepthMax$ is the maximum depth of the *is-a* tree of WordNet. $DepthMax$ in WordNet version 1.7.1 is 17.

For factor D_2, we have that if A is equal to B then D_2 is 1, otherwise it is:

$$D_2 = 1 - \frac{|D(A, MSCA) - D(B, MSCA)|}{\sqrt{D(A, MSCA)^2 + D(B, MSCA)^2}} \tag{4.30}$$

For factor D_3 we have:

$$D_3 = \frac{Depth(MSCA)}{DepthMax} \tag{4.31}$$

where $Depth(MSCA)$ is the depth of $MSCA$ in the WordNet *is-a* tree.

Similarity Metric for Mapping Relations

The previous metric is used by the object-based mapping algorithm, where the best candidate object is selected for mapping with a problem object. For the relation-based mapping algorithm, the selection metric is not used for ranking objects, but for ranking relations. The selection metric for relations is based on the object similarity metric, but takes also into account the relations, and the objects linked by the relations. The metric for ranking mapping relations is defined by the following expressions:

- Suppose that the two relations involved are R_1 (relates A with B) and R_2 (relates C with D), and:
  - MAB is the $MSCA$ between A and B (see figure 4.17 for an illustration).
  - MAC is the $MSCA$ between A and C.
  - MBD is the $MSCA$ between B and D.
  - MCD is the $MSCA$ between C and D.
- If MAB, MAC, MBD and MCD exist, Then the value for the relation similarity metric is:

  $$\omega_1 \cdot ASim(A, C) + \omega_2 \cdot ASim(B, D) + \omega_3 \cdot RASim(R_1, R_2) \tag{4.32}$$

  where $ASim(X, Y)$ is the similarity metric for mapping objects X and Y (described in the previous subsection). Weights ω_1, ω_2 and ω_3 are: 0.25, 0.25 and 0.5.

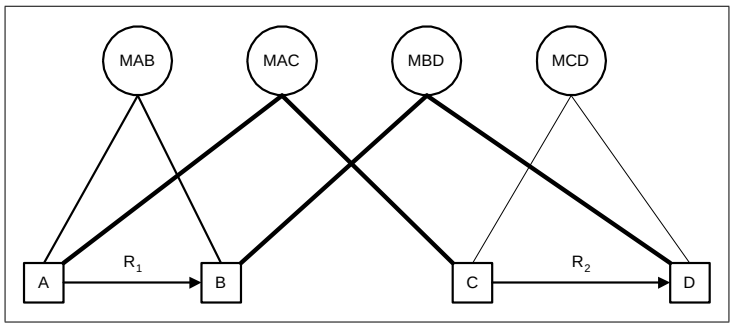


Figure 4.17: An illustration of the most specific common abstractions used in the similarity metric for mapping relations.

- If (MAC and MBD exist) and (MAB and MCD do not exist), Then the metric value is:

$$\frac{ASim(A,C) + ASim(B,D)}{2} \tag{4.33}$$

- Else the metric value is 0, meaning that the relations have no similarity.

$RASim(R_1, R_2)$ evaluates the similarity between R_1 and R_2, and is given by:

$$RASim(R_1,R_2) = \omega_1 \cdot Length(R_1,R_2) + \omega_2 \cdot Angle(R_1,R_2) + \omega_3 \cdot Depth(R_1,R_2) \tag{4.34}$$

$$Length(R_1,R_2) = 1 - \frac{|\sqrt{D(A,MAB)^2 + D(B,MAB)^2} - \sqrt{D(C,MCD)^2 + D(D,MCD)^2}|}{\sqrt{D(A,MAB)^2 + D(B,MAB)^2} + \sqrt{D(C,MCD)^2 + D(D,MCD)^2}} \tag{4.35}$$

$$Angle(R_1,R_2) = 1 - \left| \frac{D(A,MAB)}{\sqrt{D(A,MAB)^2 + D(B,MAB)^2}} - \frac{D(C,MCD)}{\sqrt{D(C,MCD)^2 + D(D,MCD)^2}} \right| \tag{4.36}$$

$$Depth(R_1,R_2) = 1 - \frac{|Depth(A) + Depth(B) - Depth(C) - Depth(D)|}{2 \cdot DepthMax} \tag{4.37}$$

where ω_1, ω_2 and ω_3 are weights with values: 0.25, 0.25 and 0.5. $Length(R_1, R_2)$ is a factor that reflects the distance between objects in the relations. It compares the distances between $A - MAB$ and $B - MAB$ with their counterparts, $C - MCD$ and $D - MCD$. $Angle(R_1, R_2)$ reflects the angle between the objects and the respective

MSCAs. Considering the trees $A - MAB - B$ and $C - MCD - D$ (see figure 4.17) it compares the disequilibrium between branches. $Depth(R_1, R_2)$ reflects the depth in the WordNet, or in other words, the abstraction level of objects. Compares the absolute concept depths in the WordNet *is-a* tree.

4.2.3 Knowledge Transfer

The last step is the generation of new diagrams using the established mappings. For each mapping the analogy module creates a new diagram, which is a copy of the query diagram. Then, using the mappings between the query objects and the candidate objects, the algorithm transfers knowledge from the candidate diagram to the new diagram. This transfer comprises two steps: first an internal object transfer, and then an external object transfer. In the internal transfer, the mapped query object gets all the attributes and methods from the candidate object that were not in the query object. This way, the query object is completed by the internal knowledge of the candidate object. The second step transfers neighbor objects and relations from the mapped candidate objects to the query objects, from the new diagram. In this process, new objects and relations are transferred to the new diagram, expanding it. If there is one object in the source diagram that is considered equivalent (same name and synset) to an object in the new diagram, then this objects are merged. The merging operation consist on matching the attributes and methods of both objects, and the merging of objects' relations. The ones in the source object that are not in the target object are copied to the target object.

4.2.4 An Analogy Example

This section presents an example of application of the analogy mechanism. The initial problem is to design an information system for an University. This problem is described in REBUILDER as an UML class diagram (see figure 4.18). The diagram is very simple, since it is described by two main objects.

The software designer can use the analogy command to perform the generation of a new diagram based on the target problem. This command first invokes the retrieval algorithm that uses the synset of the package *University* as the starting point for

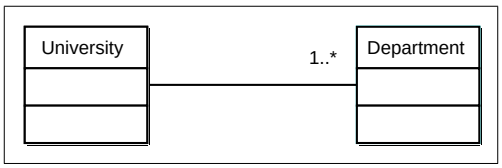


Figure 4.18: The initial class diagram used as target problem.

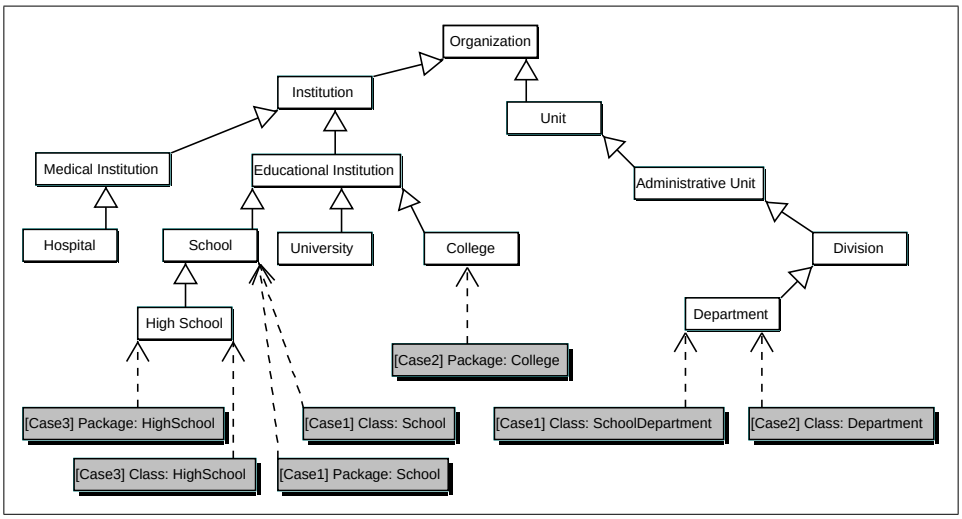


Figure 4.19: Part of WordNet structure used in the retrieval algorithm to search for similar cases. White boxes are WordNet synset nodes and grey boxes are index nodes.

searching WordNet. Figure 4.19 illustrates the application of the retrieval algorithm. The search starts at the *University* synset, which does not index packages. Supposing that the number of packages to be retrieved is three, the algorithm continues searching for index packages in a spreading activation way. In the next iteration the algorithm explores the neighborhood of the *University* synset. The only synset that is neighbor to *University* is *Educational Institution*. This synset has no indexes, so the algorithm expands the search to the next neighbor border synsets, looking for indexes. Iteration number three reaches *Institution*, *School* and *College*. From these, the algorithm collects two package indexes: one from *Case1* representing a *School*, and another from *Case2* representing a *College*. Nevertheless the number of cases is not enough, so there is one more iteration, which yields a third index package: *Case3* representing a *HighSchool*. With these three packages, the algorithm stops the exploration and initiates package ranking. Packages *Case1*, *Case2* and *Case3* are shown, respectively, in figures 4.20, 4.21, and 4.22.

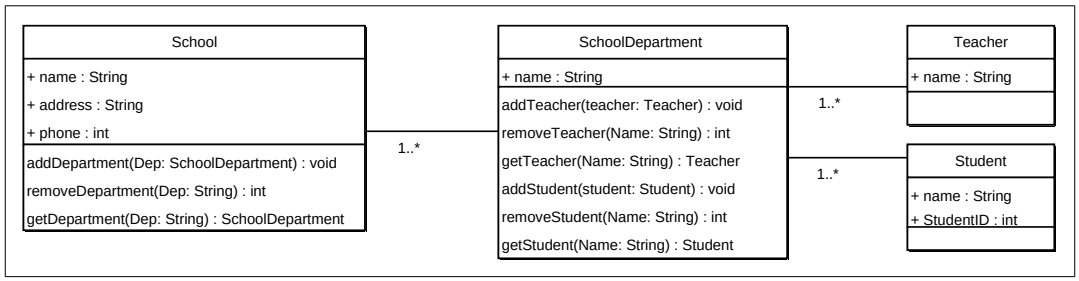


Figure 4.20: The *School* package of *Case1*.

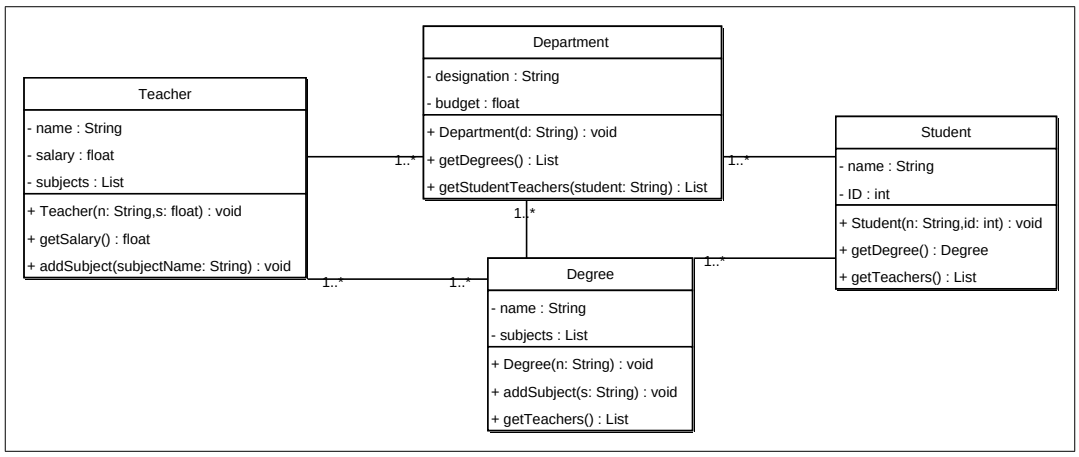


Figure 4.21: The *College* package of *Case2*.

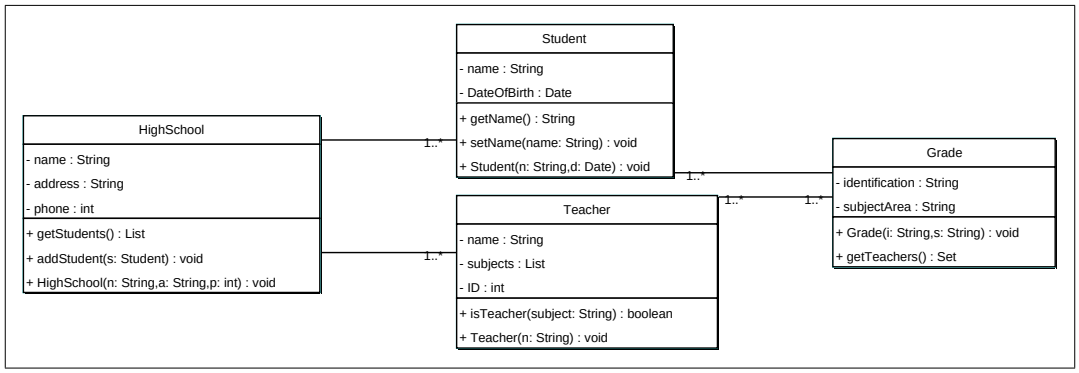


Figure 4.22: The *HighSchool* package of *Case3*.

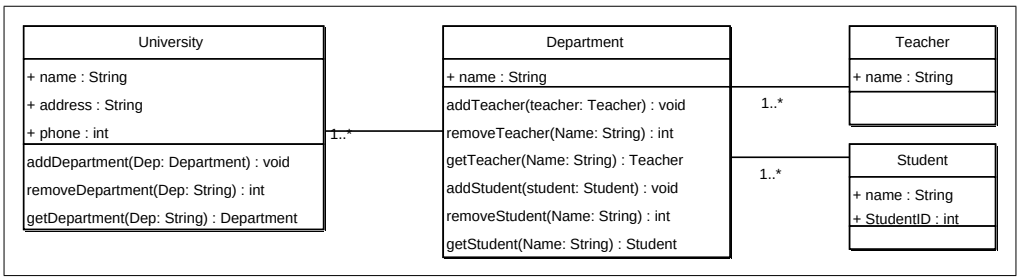


Figure 4.23: The new diagram generated by analogy with *School* package.

The package ranking is performed using the similarity metrics mentioned in subsection 4.1.2. The similarity between the synsets of packages takes into account the conceptual distance between the target synset (*University*) and the retrieved ones (*School*, *College* and *HighSchool*). As can be seen in figure 4.19 the *School* and *College* synsets are at the same distance from *University*, while the *HighSchool* synset is far away from the target synset, like the other two synsets. Taking into account the diagram similarity, the *School* package can map both problem objects (*School* with *University* and *SchoolDepartment* with *Department*). The other two packages can only map one of the problem objects. *College*: *Department* with *Department*; and *HighSchool*: *HighSchool* with *University*. Clearly the *School* package is more similar to the target problem than *College*, and *College* is more similar to the problem than *HighSchool*. This results in the following ranking of cases: first *Case1*, second *Case2*, and third *Case3*.

Using the best case that was retrieved (*Case1*), the analogy mechanism maps the problem objects to the *School* package objects. This yields the same mappings as the ones in the retrieval phase (*School* with *University* and *SchoolDepartment* with *Department*).

Then the algorithm creates a copy of the target problem (a new package), and makes the transference of knowledge between the *School* package and the new package. First the attributes and methods of *School* and *SchoolDepartment* are transferred into the mapped objects. Then the *Student* and *Teacher* classes are transferred into the new package, maintaining their relations. The final result is show in figure 4.23.

4.3 Design Composition Module

The Design Composition module generates new diagrams by decomposition and composition of cases. The input data used in the composition module is an UML class diagram, in the form of a package. This is the designer's query, which usually is a small class diagram in its early stage of development (see Figure 4.24). The goal of the composition module is to generate new diagrams that have the query objects, thus providing an evolved version of the query diagram. Generation of a new UML design using case-based composition involves two main steps: retrieving cases from the case library to be used as knowledge sources, and using the retrieved cases (or parts of them) to build new UML diagrams. In the first step, the selection of the cases to be used is performed using the retrieval algorithm and the similarity metrics described in section 4.1. The adaptation of the retrieved cases to the target problem is based on two distinct strategies: best case composition, and best complementary cases composition. The next subsections describe these two composition strategies.

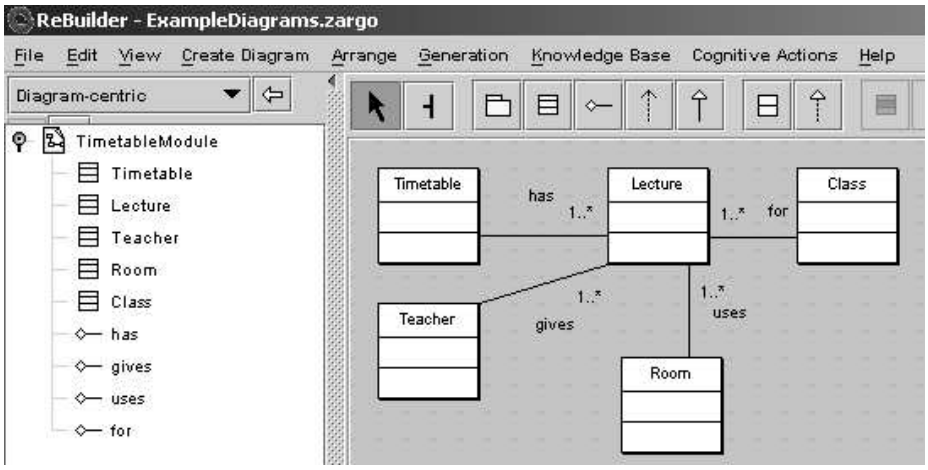


Figure 4.24: An example of an class diagram in the early stages of development.

4.3.1 Best Case Composition

The idea of the best case composition is to select the case most similar with the problem, use what this case has in common with the problem, and try to complete it using pieces of other cases.

The best case composition algorithm is shown in figure 4.25. The algorithm starts by retrieving a set of similar cases from the library (line 1). Then the retrieved

1. *RetrievedCases* ← Retrieve cases from the Case Library using *Problem*
2. *BestCase/Mapping* ← Select the best case and map it to the problem
3. *NewCase* ← Use *BestCase*, *Mapping* and *Problem* to generate a new case
4. Remove *BestCase* from *RetrievedCases*
5. WHILE *NewCase* does not map all the *Problem* objects AND *RetrievedCases* $\neq \varnothing$ DO
6. *SelectedCases* ← Search *RetrievedCases* for cases with unmapped problem objects
7. *SelectedCase* ← Select the best one, the one with more unmapped problem objects
8. *NewCase* ← Complete *NewCase* with *SelectedCase*
9. Remove *SelectedCase* from *RetrievedCases*
10. ENDWHILE
11. Return *NewCase*

Figure 4.25: The best case composition algorithm.

case most similar to the problem is mapped to the problem objects (line 2). The mapped objects in the case are transferred to a new case (line 3). If this new case maps successfully all the problem objects, then the adaptation process is finished (the while conditions in line 5). Otherwise it selects the retrieved case, which best complements the new case (in relation to the problem, see lines 6 and 7), and uses it to get the missing parts (line 8). This process continues while there are unmapped objects in the problem definition. In the end it returns the generated case (line 11).

Note that, if there are objects in the used case that are not in the problem, they can be transferred to the new case, generating new objects, but only if the mapped objects depend on them.

An UML object A (package, class or interface) depends on an object B, if there is a relation between A and B, such that:

- A is a specialization of B (there is a generalization from A to B), or
- A is a realization of B (there is a realization from A to B), or
- A depends on B (there is a dependency relation from A to B), or
- A is associated with B (there is an association between A and B).

4.3.2 Best Set of Cases Composition

The idea of this strategy is to get the best complementary set of cases, and try to build a new diagram using these cases. The algorithm for this strategy is described in figure 4.26. The best set of cases composition approach starts by retrieving and matching each retrieved case to the problem (line 1 and cycle for in lines 2 and 3),

1. *RetrievedCases* ← Retrieve cases from the Case Library using *Problem*
2. FOR*RetrievedCase* in *RetrievedCases* Do
3. *Mapping* ← Map *RetrievedCase* to *Problem*
4. ENDFOR
5. *CaseSets* ← Create the sets of complementary cases based on each case mapping
6. *BestSet* ← Select the best set from *CaseSets*, this is the set whose mapping has the best coverage of problem objects
7. *NewCase* ← Generate a new case using the *BestSet*
8. Return *NewCase*

Figure 4.26: The best set of cases composition algorithm.

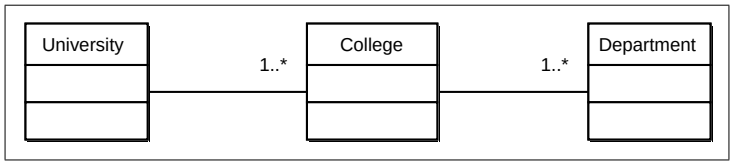


Figure 4.27: An example of a class diagram representing a problem (*P1*) .

yielding a mapping between the case objects and the problem objects. This is used to determine the degree of problem coverage of each case (line 5), after which several sets of cases are constructed. These sets are based on the combined coverage of the problem, with the goal of finding sets of cases that globally map all the problem objects. The best matching set, the one with best problem coverage (in case of a draw the sets with less elements are chosen), is then used to generate a new case (line 6 and 7). Finally it returns the generated case (line 8).

4.3.3 A Design Composition Example

This example provides an illustration of the design composition mechanism. Suppose that the class diagram of figure 4.27 is used as problem (*P1*) for design composition. The first step is to retrieve relevant diagrams and rank them. The retrieval algorithm starts by the package synset of *P1*, which is the synset corresponding to *University*. Suppose that the algorithm retrieves two cases: *Case1* corresponding to the diagram of figure 4.28, and *Case2*, which corresponds to the class diagram of figure 4.29. These cases are then ranked by similarity to the problem, which gives *Case1* a score of 0.25 and *Case2* 0.7.

The next step is to build a new case using the composition strategies. We selected the *best case composition* strategy to generate the new case. The first thing that the algorithm does is to create a new case, which is a copy of *P1*. *Case2* is selected

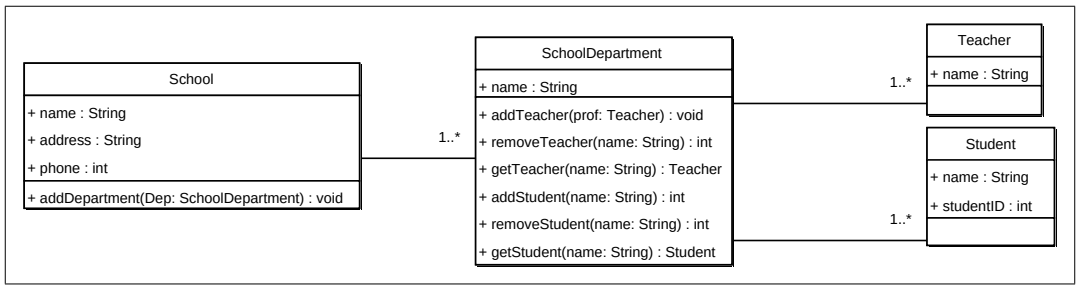


Figure 4.28: A class diagram representing part of a school information system (*Case1*).

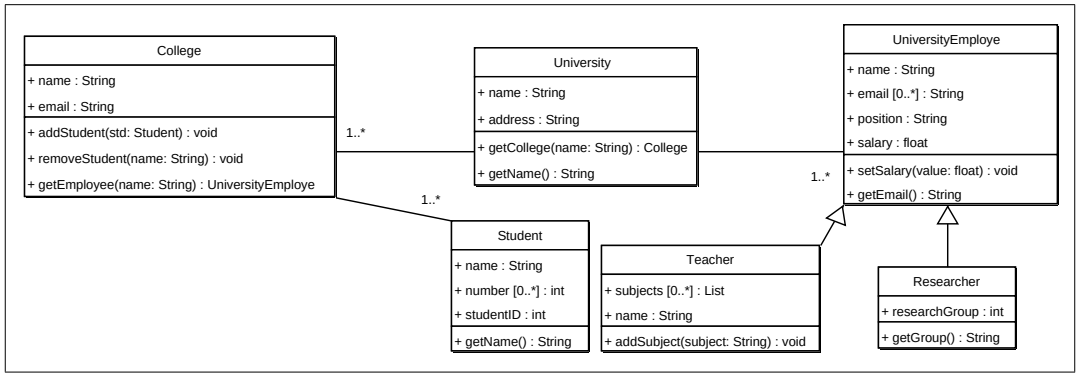


Figure 4.29: Class diagram of *Case2* (an *University*).

due to its higher similarity with the problem. The design composition algorithm maps *University* and *College* in *Case2* to *P1*, transferring the mapped case objects to the new case. Then, the objects of *Case2* that were not mapped are transferred to the new case, but only if mapped objects depend on them. After these operations, the problem has an unmapped object, which is *Department*. *Case1* has an object (*SchoolDepartment*) that has the same classification as *Department*. These two objects are mapped completing the new case. The resulting diagram is presented in figure 4.30.

Note also that all additional objects from *Case2* have been transferred to the new case because they depend on the mapped objects. Other aspect, is that the mapping of *Case1* with the problem added two additional objects (*Student* and *Teacher* from *Case1*), which already existed in the new case (from *Case2*). These two additional objects were merged with the corresponding objects yielding the new case.

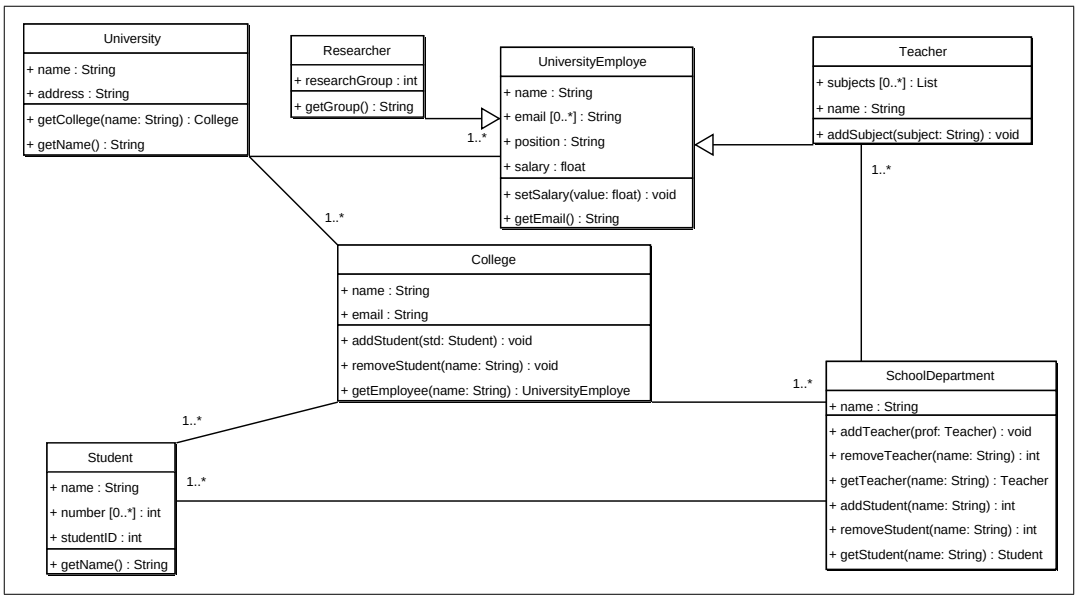


Figure 4.30: The solution generated by the design composition mechanism.

4.4 Design Pattern Module

This section presents how software design patterns can be applied to a target design using CBR. We start by describing the design patterns module, and then we present each submodule in more detail.

4.4.1 Architecture

Figure 4.31 presents the architecture of the design patterns module. It comprises three submodules: retrieval of applicable Design Pattern Application (DPA) cases, selection of the best DPA case, and application of selected DPA case. A DPA case describes the application of a specific design pattern to a software design (subsection 3.3.1 describes the case representation in detail). This module is used when the designer decides to apply design patterns to improve the current design.

The first submodule uses a target class diagram as the problem, and searches the DPA case library for DPA cases that match the problem. Then the retrieved DPA cases are ranked and the best one is selected for application, which is performed in the next step. The application of the DPA case uses the design pattern operators and yields a new class diagram, which is then used to build a new DPA case. This new

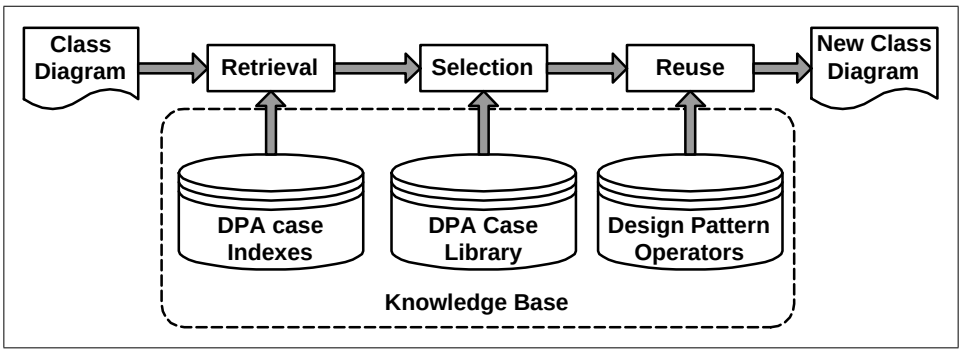


Figure 4.31: Module for software design pattern application.

Table 4.7: The participants specification for the Abstract Factory pattern operator.

Name	Type	Mandatory	Unique	Description
AbstractFactory	Object	no	yes	Declares an interface for operations that create abstract product objects.
ConcreteFactory	Object	no	no	Implements the operations to create concrete product objects.
AbstractProduct	Object	no	no	Declares an interface for a type of product object.
ConcreteProduct	Object	yes	no	Defines a product object to be created by the corresponding concrete factor.
Client	Object	no	no	Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

case is stored in the DPA case library.

4.4.2 Software Design Pattern Operators

For each design pattern there is one operator, for instance, the Abstract Factory design pattern has a specific pattern operator, which defines the application conditions of the Abstract Factory pattern and how to apply it. A software design pattern operator comprises three components: the set of specific participants, the application conditions, and the actions for the specific design pattern.

The participants are key objects, methods or attributes that play an important and active role in a design pattern. An example of the participants specification for the Abstract Factory pattern operator is described in table 4.7.

Application conditions define the constraints that must be met by participant

1. *NewClassDiagram* ← Copy *ClassDiagram*
2. *ConcreteProducts*← Get all concrete products from the mapping done between the selected case and the problem (*Role* = *ConcreteProduct*)
3. *Clients* ← Get all classes that use at least one *ConcreteProduct*
4. *AbstractFactory* ←Create new class with name "AbstractFactory"
5. Add *AbstractFactory* to *NewClassDiagram*
6. *AbstractProducts* ← For each *ConcreteProduct* get or create respective *AbstractProduct*
7. *ConcreteFactories* ← Get or create all the *ConcreteFactories* for *ConcreteProducts*
8. FOREACH *Product* in *ConcreteProducts* DO
9. *AbstractProduct* ← Get *Product* superclass
10. Abstract the access of *Clients* from *Product* to *AbstractProduct*
11. Encapsulate the construction of *Product* in *AbstractFactory* and *AbstractProduct*
12. ENDFOR
13. Apply the Singleton pattern to *AbstractFactory*
14. RETURN *NewClassDiagram*

Figure 4.32: The application algorithm for the Abstract Factory design pattern.

objects in order to apply the operator. In the case of the Abstract Factory the application conditions are: there must be at least one *ConcreteProduct*, and all *ConcreteProducts* must have no public attributes or static methods.

The pattern actions for Abstract Factory are defined in the algorithm presented in figure 4.32. This algorithm transforms *ClassDiagram* into *NewClassDiagram* through the application of the Abstract Factory design pattern.

The algorithm starts by creating a copy of the initial class diagram, and then creates all the participant objects: the *ConcreteProducts*, which have been identified in the mapping provided as input to the algorithm; the *Clients*, which are all the objects that depend or reference any *ConcreteProduct*, they can be identified in the mapping list, or the algorithm will identify them; the class *AbstractFactory* that is created by the algorithm; the *AbstractProducts*, each *ConcreteProduct* must have a correspondent *AbstractProduct*; and finally the *ConcreteFactories*, one for each *ConcreteProduct* type. Then for each *Product* in *ConcreteProducts* the algorithm abstracts the access of *Clients* from *Product* to it's superclass, and encapsulates the creation of *Product* in *AbstractFactory* and *AbstractProduct*. Applies the singleton pattern to *AbstractFactory* and returns the created diagram.

4.4.3 Retrieval of DPA Cases

The retrieval of DPA cases is performed using the WordNet as the indexing structure. The DPA case retrieval algorithm (see figure 4.33) receives the input parameters: *ClassDiagram* - the target diagram, *NumberOfCases* - number of desired solutions,

```
1. Objects ← Get all classes and interfaces from the ClassDiagram
2. Synsets ← ∅
3. FORALL Object in Objects DO
4.       Add to Synsets the Object's synset
5. ENDFOR
6. SelectedCases ← ∅
7. SearchLevel ← 0
8. Explored ← ∅
9. WHILE (#SelectedCases < NumberOfCases) AND (SearchLevel < MSL) AND
       (Synsets ≠ ∅) DO
10.      DPACases ← Get all DPA cases indexed by at least one synset from Synsets
11.      Add to SelectedCases the DPACases list
12.      NewSynsets ← ∅
13.      FORALL Synset in Synsets DO
14.          Neighbors ← Get Synset hypernyms, hyponyms, holonyms and meronyms
15.          Neighbors ← Neighbors − Synsets − Explored − NewSynsets
16.          Add to NewSynsets the Neighbors list
17.      ENDFOR
18.      Add to Explored the Synsets list
19.      Synsets ← NewSynsets
20.      SearchLevel ← SearchLevel + 1
21. ENDWHILE
22. RETURN SelectedCases
```

Figure 4.33: The retrieval algorithm for DPA cases.

and *MSL* - maximum search level for the retrieval algorithm; returning a set of retrieved cases (*SelectedCases*).

This algorithm is very similar to the case retrieval algorithm presented in subsection 4.1.1. It gets all classes and interfaces from the *ClassDiagram* into the list *Objects*, initializes the list of current (*Synsets*) to null and for each element in *Objects* gets the respective synset and adds it to *Synsets*. Basically these synsets are used as initial probes to search the WordNet structure. Three variables are initialized: the list of selected cases (*SelectedCases*) is set to empty, the variable that indicates the search level (*SearchLevel*) is set to zero, and the list of explored synsets (*Explored*) is also set to an empty list. Then the algorithm goes into a cycle while the number of selected cases is less than the number of cases to be retrieved, and the level of search in WordNet (the number of expansion iterations) is less than the *MSL*, and there are still synsets to be explored. During this cycle the algorithm performs the following steps: get all the DPA cases indexed by at least one synset in *Synsets* and add it to *SelectedCases*; initialize the list *NewSynsets* to empty and get all the adjacent synsets to the elements in *Synsets*, adding them to *NewSynsets* and removing already explored synsets; add the elements of *Synsets* to *Explored*; assign the new synset frontier (*NewSynsets*) to *Synsets*; and increment *SearchLevel*. At the end of the while cycle the algorithm returns the found cases (*SelectedCases*). The algorithm expands the search to neighbor synsets using all the WordNet semantic relations (*is-a*, *part-of*,

member-of, and *substance-of*), performing an exhaustive search of the WordNet.

4.4.4 Selection of DPA Cases

After retrieval of the relevant cases, they are ranked accordingly to their applicability to the target diagram (*ClassDiagram*).

The selection algorithm (see figure 4.34) starts by initializing the list of scores (*Scores*) and mappings (*Mappings*), in lines 1 and 2. Then maps the *ClassDiagram* with each of the retrieved cases (*SelectedCases*), resulting in a mapping for each case (lines 3 to 7). The mapping is performed from the case's participants to the target class diagram (only the mandatory participants are mapped). Associated to each mapping there is a score, which is given by the number of mapped participants. What this score measures is the degree of participants mapping between the DPA case and the target diagram. The score of a mapping (M) is computed using the following equation:

$$Score(M) = \omega_1 \cdot \frac{TScoreObj}{CObjs} + \omega_2 \cdot \frac{Methods}{CMets} + \omega_3 \cdot \frac{Attributes}{CAttrs} \tag{4.38}$$

where $TScoreObj$ is the score of the object mapping participants, which is computed by summing the WordNet distance between mapped objects. $Methods$ and $Attributes$ are, respectively, the number of mapped method participants and mapped attribute participants. $CObjs$, $CMets$ and $CAttrs$ are, respectively, the number of object participants in the selected case, the number of method participants in the selected case, and the number of attribute participants in the selected case. ω_1, ω_2 and ω_3 are constants with values 0.5, 0.25 and 0.25.

The next step in the algorithm is to rank the *SelectedCases* list based on the mapping scores (line 8). The final phase consists on checking the applicability of the best DPA case (lines 9 to 13), which is performed using the design pattern operator associated with the DPA case. If the application conditions of this operator are not violated, then this DPA case is returned as the selected one (line 11). Otherwise, this case is discarded and the next best case goes through the same process, until one applicable case is found or it returns null (line 14).

```
1. Scores ← ∅
2. Mappings ← ∅
3. FORALL SelectedCase in SelectedCases DO
4.       Mapping/Score← Get the mapping and score for the SelectedCase
5.       Add to Mappings the SelectedCaseMapping
6.       Add to Scores the SelectedCaseScore
7.ENDFOR
8.Rank lists: SelectedCases, Mappings and Scores, by Scores
9.FORALL SelectedCase in SelectedCases DO
10.    IF (DesignPattern(solution of SelectedCase) can be applied to ClassDiagram
                              using the Mapping established before) THEN
11.       RETURN SelectedCase and the respective Mapping
12.    ENDIF
13. ENDFOR
14. RETURN NULL
```

Figure 4.34: The algorithm for selection of DPA cases.

4.4.5 Application of DPA Cases

Once selected the DPA case, the next step is to apply it to the target class diagram generating a new class diagram and a new DPA case. The solution of the selected DPA case is a design pattern. This design pattern has a corresponding pattern operator, which will be applied to the target class diagram. Starting with the participants mapping established before, the application of the pattern is performed using the application algorithm of the pattern operator.

4.4.6 An Application Example

This section presents an example for illustration of the functioning of the design pattern module. Figure 4.35 shows the target class diagram used in our example (class attributes and methods are omitted for the sake of simplicity). The goal is to improve this class diagram through the application of design patterns. In the remaining of this section we will exemplify the three phases of the design pattern application module: retrieval, selection and application of DPA cases.

The first step is the retrieval of DPA cases from the case library. This is performed using the algorithm presented in this section. The initial search probes are the synsets associated to: *Repair Shop*, *Stock*, *Motor* (the synsets for *Car Motor* and *Motorcycle Motor* are the same of *Motor*), and *Wheel* (the synsets for *Car Wheel* and *Motorcycle Wheel* are the same of *Wheel*). The case library comprises several DPA cases, from which we present the initial class diagrams for two of them (see figures 4.36 and 4.37). DPA case 1 represents the application of the Abstract Factory design pattern

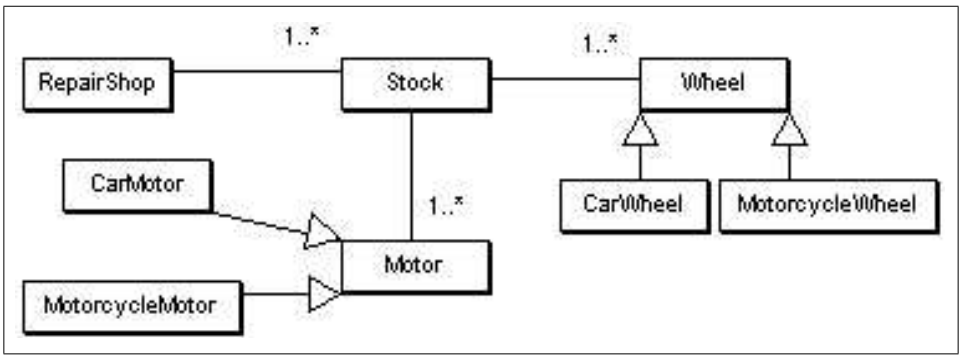


Figure 4.35: Target class diagram used in the example.

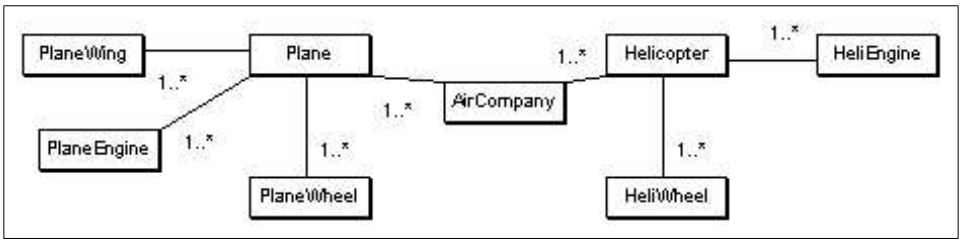


Figure 4.36: The initial class diagram of DPA case 1.

to the initial class diagram of figure 4.36. The participants in this DPA case are: *PlaneEngine*, *PlaneWheel*, *HeliEngine* and *HeliWheel* which are Concrete Products; and *Plane* and *Helicopter* which are Clients. The solution of this DPA case is Abstract Factory.

The DPA case 2 is also the representation of an application of the Abstract Factory pattern, but to the initial class diagram of figure 4.37. The participants are: *TTLLogicGate*, *TTLMultiplexer*, *TTLDecoder*, *CMOSLogicGate*, *CMOSMultiplexer*, and *CMOSDecoder*, which are Concrete Products; and *Equipment* which is a Client.

The search then starts using these four synsets and is performed in the WordNet

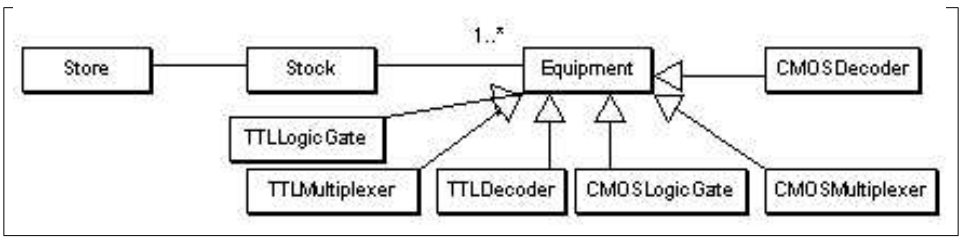


Figure 4.37: The initial class diagram of DPA case 2.

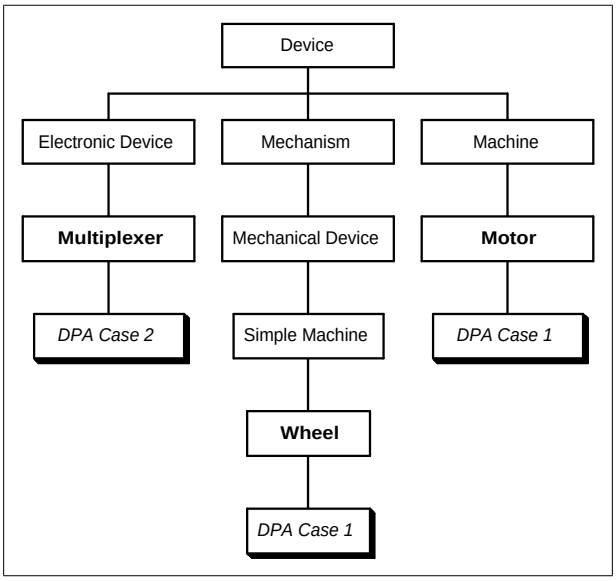


Figure 4.38: Part of the WordNet *is-a* structure used in the retrieval example.

structure. Figure 4.38 presents part of the WordNet structure, which indexes DPA case 1 (through the synset of *Wheel* and *Motor*, *Engine* has the same synset as *Motor*) and DPA case 2 (through the synset of *Multiplexer*). Starting by *Wheel* and *Motor*, case 1 is the first being selected, then the algorithm expands to the neighbor nodes until the fourth iteration, when it gets to *Multiplexer* (*Motor* - *Machine* - *Device* - *Electronic Device* - *Multiplexer*) where it founds case 2. If we assume that only two cases are needed, the cases retrieved for the next step are: DPA case 1, and DPA case 2.

The next step is to select which of the retrieved cases is going to be applied. The first phase of selection establishes the mappings and the score associated with each mapping. For case 1 the following mappings were obtained:

- *PlaneEngine* - *CarMotor*
- *PlaneWheel* - *CarWheel*
- *HeliEngine* - *MotorcycleMotor*
- *HeliWheel* - *MotorcycleWheel*

Mappings for case 2 are:

- *TTLLogicGate* - *Motor*

- *TTLMultiplexer - CarMotor*
- *TTLDecoder - CarWheel*
- *CMOSLogicGate - Wheel*
- *CMOSMultiplexer - MotorcycleMotor*
- *CMOSDecoder - MotorcycleWheel*

Mappings for case 1 get the following score: 0.5 * (4 / 4) + 0.25 * 1 + 0.25 * 1 = 1, according to equation 4.38. Notice that all the mappings established between case 1 and the target diagram match perfectly, so they all have score 1, yielding 4 for the sum. Since there are no method or attribute participants, the score associated with each of these items is 1. For case 2, the computation of the mappings' score is: 0.5 * ((0.31 * 4 + 0.34 * 2) / 6) + 0.25 * 1 + 0.25 * 1 = 0.66. There are six mappings between case 2 and the target diagram, which leads to the following semantic distances: *LogicGate* to *Motor*, *Decoder* to *Wheel*, *LogicGate* to *Wheel*, has 8 *is-a* semantic relations, yielding the similarity of 0.31 (see equation 4.4); and *Multiplexer* to *Motor* has 6 *is-a* semantic relations, yielding the similarity of 0.34.

To this point, we have case 1 ranked in the first place. The next step is to assess the applicability of the solution pattern of case 1, which is the Abstract Factory pattern. The pre-conditions necessary for this pattern to be applied are:

- Concrete Products must have at least one element.
- All classes in Concrete Products must:
  - exist, and
  - have no public attributes, and
  - have no static methods.

Assuming that the target diagram meets all the pre-conditions, the Abstract Factory can be applied, using the mapping established before.

The final step is the application of the selected design pattern, using the respective design pattern operator. Figure 4.32 presents the application algorithm for the

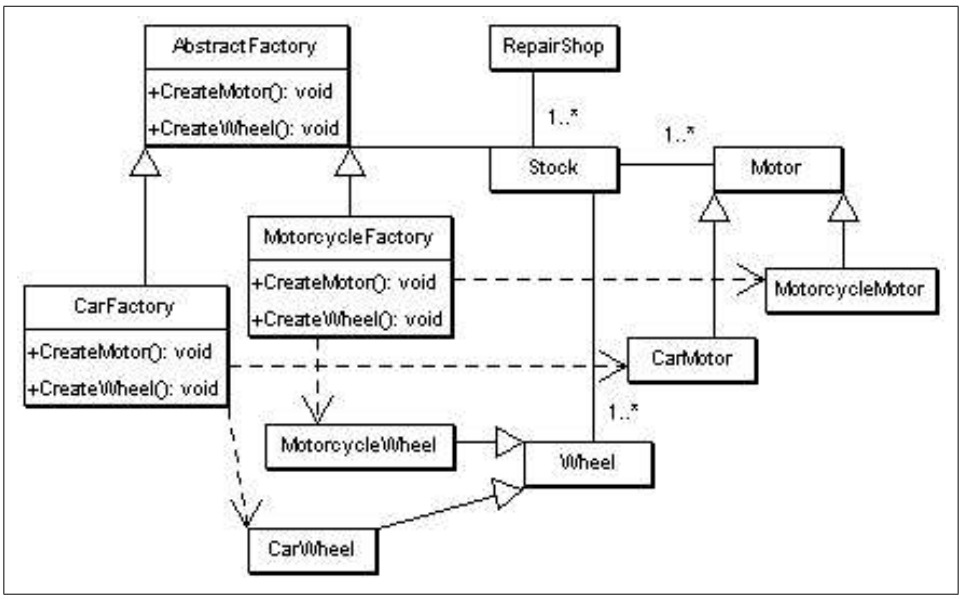


Figure 4.39: The new class diagram generated from the application of DPA Case 1 to the target diagram.

Abstract Factory design pattern, which after being applied to the class diagram of figure 4.35 (using the mapping determined before) results in the diagram of figure 4.39. One of the advantages of this transformation, is that, it is easier to maintain and extend this design. Because it enables the extension of the design in two different dimensions: types of products that can be added (for instance steering mechanism), family of products (for instance truck).

4.5 Verification and Evaluation Module

The verification and evaluation module comprises two distinct phases: verifying a class diagram, and evaluating it. The next subsections describe each one of these functionalities.

4.5.1 Verification

The main idea of verification in REBUILDER is to identify errors in class diagrams applying four types of knowledge sources: design cases, WordNet, verification cases, and the designer.

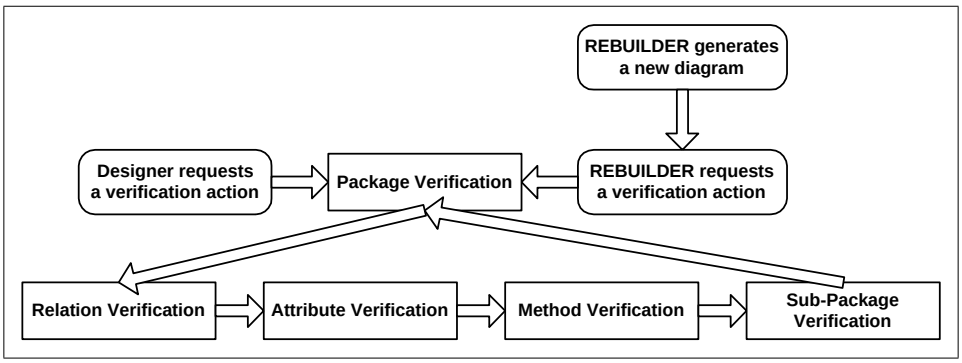


Figure 4.40: The verification process in abstract.

Not only REBUILDER generated diagrams can be verified, but also designer-created diagrams (see figure 4.40 for an abstract description of the verification process). The designer has only to select the diagram to be checked and activate the verification command to perform it. The designer can also select an option in the settings menu for automatic verification of each solution generated by REBUILDER. Verification is performed on five diagram components: packages, object names, relations, attributes and methods. Verifying a diagram component consists on determining it's validity (true or false), which is to say, if the object is correct or if it is invalid or incoherent within the diagram. In case of being correct it stays in the diagram, otherwise the object can be deleted from the diagram.

An important implementation aspect is that verification cases (which are cases that represent success or failure verification situations) are stored locally in the designer client. Thus, each designer has her/his library of verification cases, which personalizes the system. This is important, since each software designer has a different way of modelling the problem being addressed. Starting from this assumption we decided that the case base of verification cases would be local to each client (or designer). This assumption can be in the origin of some limitations when dealing with development teams. Notwithstanding, we still defend our point of view by saying that, from our experience, most of the times team development at design level is performed by one or two individuals, which then discuss their designs with the remaining of the team. Also, only one or two project members are responsible for using the CASE tool in modelling the system. Nevertheless, there are at least two ways of addressing this problem. Centralizing all the verification cases in one case base accessible to all members of the team, which would imply the maintenance of

such a case base, or impose project-wide agreements, which are always difficult to make. So, both solutions have pros and cons. In the next subsections we describe the different verification phases.

Package Verification

The package verification checks a class diagram starting with the diagram relations, then checking each object's attributes and methods, finally checking the sub-packages, recursively calling the package verification functionality.

Name Verification

Name checking applies only to packages, classes and interfaces, and is performed when REBUILDER needs to assign a synset to an object. REBUILDER reasoning mechanisms depend on the correct assignment of synsets to objects, which makes this verification very important. REBUILDER makes a morphological and compositional analysis of the object's name, trying to match it to WordNet words. The disambiguation starts by extracting from WordNet the synsets correspondent to the object's name. This requires the system to parse the object's name, which most of the times is a composition of words. REBUILDER uses specific heuristics to choose which word to use. For instance, only words corresponding to nouns are selected, because usually objects correspond to entities. A morphological analysis must also be performed, extracting the regular noun from the word. After this, a word or a composition of words has been identified and will be searched in WordNet. The result from this search is a set of synsets. From this set of synsets, REBUILDER uses the disambiguation algorithm to select one synset, which is supposed to be the correct one. Once it finds a match it can easily get the list of possible synsets for the given name. Now, two things can happen: the designer selects the correct synset, or REBUILDER uses a word sense disambiguation method to do it (it is up to the designer to decide).

Relation Verification

Relation checking is based on WordNet, design cases, and relation verification cases, which are cases describing successful and failure situations in checking a relation validity. These cases are described by:

- Type of Relation {association, generalization, realization, dependency}.
- Multiplicity {1-1, 1-N, N-N}.
- Source Object Name.
- Source Object Synset.
- Destination Object Name.
- Destination Object Synset.
- Outcome {Success or Failure}.

Two relation verification cases c_1 and c_2 match if:

$$\begin{aligned} &RelationType(c_1) = RelationType(c_2) \wedge Multiplicity(c_1) = Multiplicity(c_2) \wedge \\ &SourceObject(c_1) \equiv SourceObject(c_2) \wedge DestObject(c_1) \equiv DestObject(c_2) \end{aligned} \tag{4.39}$$

where *RelationType(c)*, *Multiplicity(c)*, *SourceObject(c)* and *DestObject(c)* are, respectively, relation type, multiplicity, source object, and destination object of verification case c. Two objects, o_1 and o_2 are said to be equivalent if:

$$\begin{aligned} o_1 \equiv o_2 \Leftrightarrow &\left[Synset(o_1) \neq \varnothing \wedge Synset(o_1) \neq \varnothing \wedge Synset(o_1) = Synset(o_1)\right] \vee \\ &\Big[\left[Synset(o_1) = \varnothing \vee Synset(o_2) = \varnothing\right] \wedge Name(o_1) = Name(o_2)\Big] \end{aligned} \tag{4.40}$$

where *Synset(o)* gives the synset of object o and *Name(o)* yields the name of object o.

All the knowledge sources are used for validating the relation being inspected. The relation verification algorithm is detailed in figure 4.41.

The algorithm starts by searching a verification case that matches the relation being verified. If a successful verification case is found, then the relation is considered valid and exits. If a case is found but it has a failure outcome, then the relation is considered invalid and it is removed from the diagram. Otherwise the algorithm

```
1. Search for an equivalent verification case in the library.
2. IF found and outcome is Success THEN
3.     Consider the relation valid and exit.
4. IF found and outcome is Failure THEN
5.     Consider the relation invalid and delete it from the diagram, and exit.
6. IF not found THEN
7.     Continue.
8. Search for an equivalent relation in WordNet.
9. IF found THEN
10.    Consider the relation valid, add a new successful verification case and exit.
11. ELSE
12.    Continue.
13. Search for an equivalent relation in the design cases.
14. IF found THEN
15.    Consider the relation valid, add a new successful verification case and exit.
16. ELSE
17.    Continue.
18. Ask the designer the relation validity.
19. IF designer considers relation valid THEN
20.    Consider the relation valid, add a new successful verification case and exit.
21. Consider the relation invalid, add a new failure verification case and exit.
```

Figure 4.41: The relation verification algorithm.

searches other knowledge sources. Next the algorithm searches for a similar relation in WordNet, if it founds one, then the relation is considered valid and exits. Otherwise, the design cases are searched for similar relations that could validate the relation being verified. If the relation's validity could not be determined then the algorithm asks the designer about the relation's validity.

A WordNet equivalent relation is a relation between two synsets in which one of them is the source object synset and the other is the destination object synset. A design case comprises an equivalent relation if it has two objects connected by a similar relation to the one being investigated, and with equivalent source objects and destination objects. Retrieval of verification cases is based on two steps: first on the relation type, and then by source object name. If there is more than one equivalent case, the outcome that is common to more cases is chosen as the correct outcome. In case of a draw, the system retrieves the newest case.

Attribute Verification

Attribute checking is based on WordNet, design cases, and attribute verification cases, which are cases describing successful and failure situations in checking an attribute validity. These cases have the following description:

- Object Name.

- Object Synset.
- Attribute Name.
- Outcome {Success or Failure}.

Two attribute verification cases vc_1 and vc_2 match if:

- The Object's names and synsets are the same, and attributes' names are also the same, or
- one of the synsets is empty and their objects' names and attributes' names are the same.

As in relation verification, all the knowledge sources are used for validating the attribute being inspected. The order of search and the algorithm that is used are the same as in relation verification (adapted to the attribute situation). A WordNet equivalent attribute is represented by a *substance-of*, *member-of* or *part-of* relation between the synset of the object being inspected and every possible synset of the attribute's name. A design case comprises a similar attribute, if there is an object with the same synset and name comprising an attribute with the same name as the attribute being inspected.

Retrieval of attribute verification cases is based on two steps. First the algorithm searches the verification cases based on object name, searching verification cases with the same object name. Then from these cases the algorithm selects only those with the same object synset. If there are more than one case in this situation, then the algorithm assesses the attribute as valid if there are more successful than failure cases, or invalid otherwise. In case of equal number of cases, then the algorithm does not decide about the attribute's validity and asks the designer. The way cases are indexed in the verification case library is depicted in figure 4.42. As can be seen, verification cases are indexed by name and by synset, which enables a verification case to be retrieved by the object (or objects) involved in the verification process.

Method Verification

Method verification is similar to the attribute verification with the exception that WordNet is not used as a knowledge source, being replaced by an heuristic rule. The

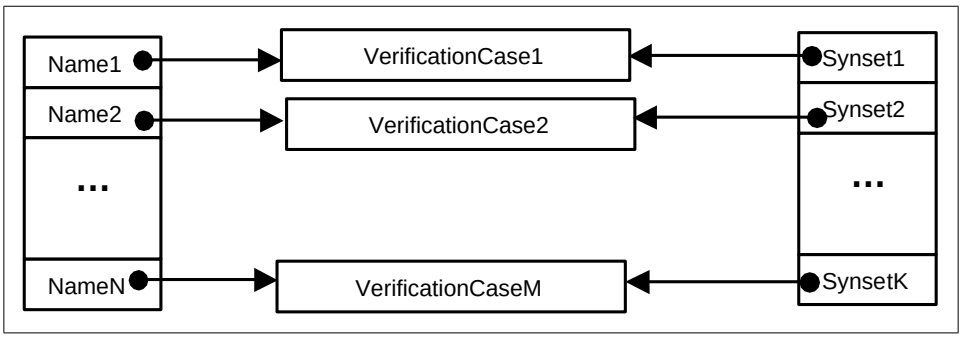


Figure 4.42: The indexing of verification cases (attribute and method).

heuristic used is: if the method name has a word that is an attribute name or a neighbor class name, then the method is considered valid. Method verification cases describe successful and failure situations in checking a method validity. These cases have the following description:

- Object Name.
- Object Synset.
- Method Name.
- Outcome {Success or Failure}.

Two method verification cases c_1 and c_2 match if:

- The objects' names and synsets, and methods' names are the same,or
- one of the synsets is empty and their names and methods' names are the same.

As in relation verification, all the knowledge sources are used for validating the method being inspected. The order of search and the algorithm used is the same as in relation verification (adapted to the method situation and with WordNet replaced by the heuristic that was referred before). A design case comprises a similar method, if there is an object with the same synset and name comprising a method with the same name as the method being inspected. Retrieval of method verification cases is performed in the same way as in attribute verification cases.

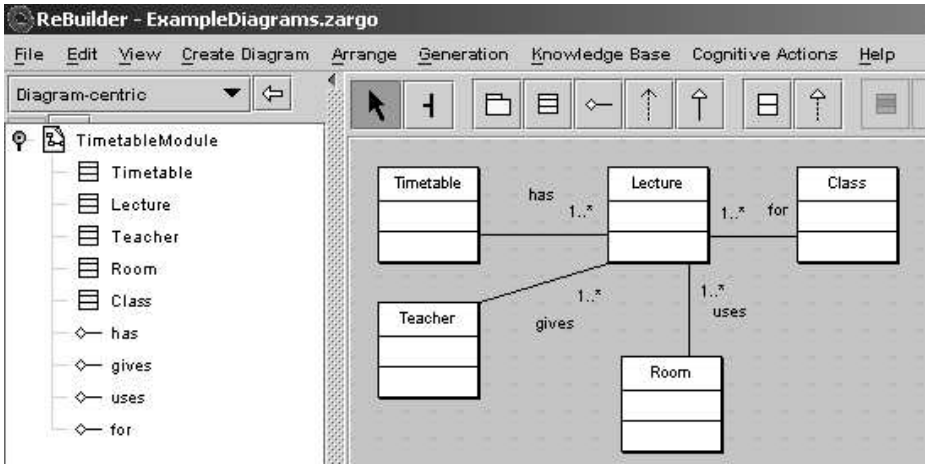


Figure 4.43: The UML class diagram used as problem for the verification example.

4.5.2 A Verification Example

This subsection illustrates the verification with an example used before. Suppose that the designer uses the diagram presented in figure 4.43 as a problem for the design composition module, resulting in a new diagram. Part of this diagram is presented in figure 4.44, which shows some inconsistencies. For instance: the generalization between *Teacher* and *Timetable*, or the method *addStudent* in *Timetable*, or the attribute *studentID* in class *Lecture*.

The verification process starts by checking the package containing the new diagram. For illustration purposes we will describe the process by checking only the diagram elements considered invalid, the ones mentioned before. When the verification process reaches the generalization between *Timetable* and *Teacher*, the system first searches in the verification cases for an equivalent case, suppose that it does not exist. Then it searches in WordNet for an *is-a* relation between the *Timetable* synset and *Teacher* synset, suppose also that it does not exist. Then searches the design cases that do not have any similar relation, and finally asks the designer for a validation on this relation, which s/he answers as invalid. Then the system adds a new relation verification case comprising: [Generalization, 1-1, Teacher, 108756476[7], Timetable, 105441050, Failure], and deletes the relation. The next time the system finds an identical relation it will be considered invalid and will be deleted from the design. Now suppose the system is checking the *studentID* attribute of class *Lecture*. The attribute verification process will search the case library of attribute verification

[7]Synsets are identified by nine digit numbers.

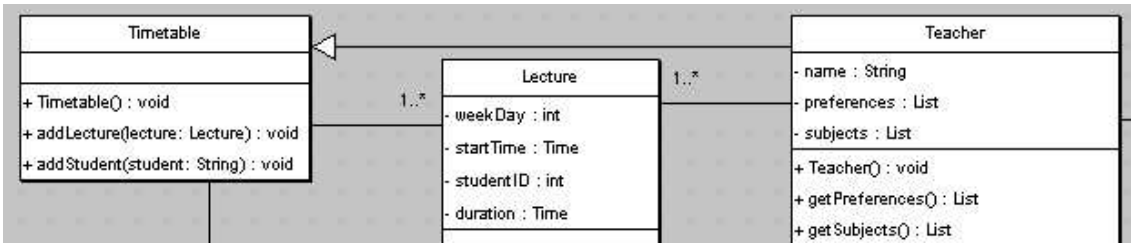


Figure 4.44: Part of the class diagram resulting from design composition.

cases and it finds an equivalent case: [MathLecture, 106035418, studentID, Failure]. The system considers the attribute invalid and deletes it from the diagram. Finally the system checks method *addStudent* from class *Timetable* and the method checking heuristic does not applies - the design cases have no similar method - then it asks the designer, which considers the method invalid. The method verification algorithm will delete the method and add a new method verification case: [Timetable, 105441050, addStudent, Failure].

4.5.3 Evaluation

The evaluation phase provides an assessment of the design characteristics based on software engineering metrics. It provides the designer with some properties of the design diagram. They are divided in two categories: software metrics for object oriented designs, and design statistics. The software metrics used are:

- Class/Interface/Package complexity (class complexity is defined as the sum of the number of methods and attributes; interface complexity is the number of attributes; package complexity is the sum of complexity of the objects it comprises ,sub-packages, classes and interfaces).
- Number of children of a Class/Interface.
- Depth of inheritance tree of a Class.
- Coupling between Classes/Interfaces: number of non-inheritance related couples of an object with other objects.
- Average of Class/Interface number of children by package.
- Average of Class/Interface depth of the inheritance tree by package.

Design statistics are (by package):

- Number of sub packages.
- Number of relations.
- Number of classes.
- Number of interfaces.
- Average number of classes by package.
- Average number of interfaces by package.
- Average number of relations by package.
- Average number of attributes by class.
- Average number of methods by class.
- Average number of methods by interface.

4.6 Learning Module

Learning in REBUILDER is performed by storing new cases in the case library. Design cases can be a result of the software designer work, or they can be the result of the CBR process. In both situations, it is up to the system's administrator to accept (or not) the case as a valid one and add it (or not) to the case library.

Several Case Base Maintenance (CBM) strategies have been implemented in REBUILDER. They provide guidance to the software designer, whether a case should be added (or not) to the case base. Most of these criteria were inspired in approaches developed for cases represented as vectors of attribute/value pairs. Since we have a complex case representation we had to adapt these strategies to our case representation. So, our contribution to this field, concerns the adaptation of these strategies to a complex case representation.

In REBUILDER, the learning mechanism is a tool at the disposal of the KB administrator. When the software designer thinks that a design is worth being integrated in the design repository (the case library) s/he can submit it. This action

sends the design to the list of unconfirmed cases in the case library. The KB administrator can then call the *Activate Learning* command to start the CBM process. For each submitted case, the learning module runs the CBM strategies that were selected by the KB administrator. These strategies provide an advice about adding (or not) submitted cases to the case library. It also gives advice about the deletion of cases in the case base. The final decision is always up to the KB administrator. The next subsections describe the CBM strategies integrated in REBUILDER and how they were adapted to our case representation.

4.6.1 Frequency Deletion Criteria

The frequency deletion criteria developed by Minton [Minton, 1990], suggests cases for deletion based on the frequency of case access. This implies the existence of an access counter associated with each case. The counter starts with zero and is incremented each time the case is retrieved. A maximum number of cases is established for the case library (called swamping limit). When the learning mechanism is activated, if the number of cases in the library reaches the swamping limit, the less retrieved cases (enough to keep the swamping limit) are suggested for deletion. If the swamping limit is not reached, new cases can be added to the case library with their frequency initialized to zero. In case there are one or more cases with the same frequency of use, then the one with the oldest access is suggested for deletion.

In REBUILDER, if the number of retrieval operations criteria is not enough, then the smallest case is suggested for deletion.

4.6.2 Subsumption Criteria

The subsumption criteria developed by Racine [Racine and Yang, 1997] defines that a case is redundant if: is equal to another case, or is equivalent to another case, or is subsumed by another case.

When a new case is to be added to the case library it must be checked for redundancy. If the case is considered redundant, then it is not added to the case library. Cases are redundant when they are subsumed by other cases. In REBUILDER a case C_1 is considered subsumed by case C_2 if the root package of C_1 is subsumed by a

package of C_2. A package Pk_1 is subsumed by a package Pk_2 if: Pk_1 and Pk_2 have the same synset, and all the diagram objects of Pk_1 (except sub-packages) have an equivalent in Pk_2, and all sub-packages in Pk_1 are subsumed by a sub-package of Pk_2. This definition is recursive, which is adequate to the tree-like structure of class diagrams. This process can be time consuming, especially if we are dealing with long design cases. Subsumed cases are suggested for deletion.

4.6.3 Footprint Deletion Criteria

Smyth and Keane [Smyth and Keane, 1995b] developed the footprint deletion criteria, which involves two important notions: coverage and reachability. Coverage of a case is considered to be the neighborhood of the case within certain adaptation limits. In other words, coverage relates to the set of problems that a case can solve.

$$CoverageSet(c \in C) = \{c' \in C : Solves(c, c')\} \tag{4.41}$$

where C is the case base considered, and c and c' are cases. The reachability set of a case, is the set of cases that can solve this case.

$$ReachabilitySet(c \in C) = \{c' \in C : Solves(c', c)\} \tag{4.42}$$

Since a target problem in REBUILDER is a class diagram, we say that a case c solves a problem p, when c subsumes p, which means:

$$Solves(c, p) = Subsumes(c, p) \tag{4.43}$$

Using these two concepts, Smyth and Keane divide cases in the case library into four types: pivotal, auxiliary, spanning, and support cases. Pivotal cases represent unique ways to answer a specific query. Auxiliary cases are those which are completely subsumed by other cases in the case base. Spanning cases are cases between pivotal and auxiliary cases, which link together areas covered by other cases. Support cases exist in groups to support an idea. The recommended order of deletion is: auxiliary, support, spanning, and pivotal cases.

In REBUILDER support cases can not be distinguished from spanning cases, due to the definitions used for the coverage and reachability sets. So we decided not to

consider support cases. The formal definitions used in REBUILDER for these types of case are:

$$\begin{aligned} Pivotal(c) &\Leftarrow \forall c' \in C : \neg Subsumes(c', c) \\ Spanning(c) &\Leftarrow \exists c' \in C : Subsumes(c', c) \wedge \exists c'' \in C : Subsumes(c, c'') \\ Auxiliary(c) &\Leftarrow \exists c' \in C : Subsumes(c', c) \wedge \forall c'' \in C : \neg Subsumes(c, c'') \end{aligned} \quad (4.44)$$

In presence of a draw, the similarity between the candidate cases and the new case is used to suggest the cases to be deleted (see subsection 4.1.2), in which the cases most similar to the new case are suggested for deletion.

4.6.4 Footprint-Utility Deletion Criteria

This criteria, also developed by Smyth and Keane [Smyth and Keane, 1995b], is the same as the Footprint Deletion Criteria, with the difference that when there is a draw the selection is based on the case usage - less used cases are suggested for deletion.

4.6.5 Coverage Criteria

The coverage criteria, as it was first devised by Smyth [Smyth and McKenna, 1998], involves three factors: case base size, case base density, and case base distribution. The competence of a case base is strongly influenced by its size, this is an intuitive statement. Another relevant factor is case base density, with the local density of a case c within a group of cases G in the case base C being defined by:

$$CaseDensity(c, G) = \frac{\sum_{c' \in G - \{c\}} Sim(c, c')}{|G| - 1} \quad (4.45)$$

where *Sim(c,c')* returns the case similarity between c and c'. The third factor that influences a case base competence is case distribution. This factor is more complex than previous factors, because if the CBR system performs adaptation and verification of solutions, this will also influence the case base distribution (besides retrieval, of course). To assess the competence of a case base, Smyth and McKenna compute the local case coverage sets and determine how these sets combine and interact to form the case base competence. They define a competence group as a set of cases which are related to each other, and that make a contribution to the case base competence,

which is independent from other competence groups. The definition of competence group is based on the shared coverage concept. Two cases exhibit shared coverage if their coverage sets overlap:

$$SharedCoverage(c, c') = true \Leftarrow CoverageSet(c) \cap CoverageSet(c') \neq \varnothing \quad (4.46)$$

A competence group can be defined as:

$$\begin{aligned} CompetenceGroup(G) = \{\forall c_i \in G, \exists c_j \in G - \{c_i\} : SharedCoverage(c_i, c_j) = true\} \\ \wedge \{\forall c_k \in C - G, \neg\exists c_l \in G : SharedCoverage(c_k, c_l) = true\} \end{aligned} \quad (4.47)$$

Smyth and McKenna show that according to this definition a case belongs only to one competence group. Competence group size and number depends on four factors: distribution of cases, density of cases, retrieval mechanism, and adaptation mechanism. Group coverage can be defined by the number and density of cases in the group. The number of cases in the group is easy to measure. Group density is given by:

$$GroupDensity(G) = \frac{\sum_{c \in G} CaseDensity(c, G)}{|G|} \quad (4.48)$$

Group coverage is based on group size and group density, and is defined as:

$$GroupCoverage(G) = 1 + [|G| \bullet (1 - GroupDensity(G))] \quad (4.49)$$

The total coverage of a case base comprising several competence groups is given by the following formula ($G = \{G_1, ..., G_n\}$):

$$Coverage(G) = \sum_{G_i \in G} GroupCoverage(G_i) \quad (4.50)$$

The learning module of REBUILDER uses these definitions to decide which cases should be suggested for deletion. A new case is added to the case library if the swamping limit has not been reached or its inclusion increases the ratio case base coverage/case base number. Otherwise the KB administrator is advised to delete the case.

4.6.6 Case-Addition Criteria

The case-addition criteria [Zhu and Yang, 1999] involves the notion of case neighborhood, which in REBUILDER is defined as:

$$Neighborhood(c) = \{c' \in C, \tau \in [0, 1] : (RPSynset(c) = RPSynset(c')) \wedge (Sim(c, c') > \tau)\} \quad (4.51)$$

1. Determine the neighborhood for every case in the case base.
2. Set S to $\varnothing$.
3. Selecta case from $C - S$ with the minimal benefit with respect to the neighborhood of S and add it to S.
4. Repeat step 3 until $Neighborhood(C) - Neighborhood(S)$ is empty or S has k elements.

Figure 4.45: The case deletion algorithm used by the case-addition criteria.

where $RPSynset(c)$ is the root package synset of case c, and τ is a threshold value used to define a case neighborhood. This criteria uses also the notion of benefit of a case c in relation to a case set S, which we define as:

$$Benefit(c, S) = \sum_{c' \in Neighborhood(c) - Neighborhood(S)} P(c') \tag{4.52}$$

The neighborhood of a set of cases is given by:

$$Neighborhood(S) = \bigcup_{c \in S} Neighborhood(c) \tag{4.53}$$

In our implementation of this criteria we defined $P(c)$ as the frequency function of case c, which is computed using an access counter associated with each case in the case library. Then the algorithm in figure 4.45 is used to determine a set of cases S defining the optimal case base coverage.

At the end, the cases in S should remain in the case base. All the others should be removed. To determine if a case that is not in the case base, should be added to the case base, just add it to the case base and run the algorithm. At the end, if the case is in S then it should be added to the case base, otherwise it's deletion is suggested.

4.6.7 Relative Coverage and Condensed NN Criteria

Smyth and McKenna [Smyth and McKenna, 1999] developed this criteria with the goal of maximizing coverage while minimizing case base size. The proposed technique for building case bases is to use the Condensed Nearest Neighbor (CNN[8], see [Hart, 1968]) on cases that have first been arranged in descending order of their relative coverage contributions. The relative coverage is defined as:

$$RelativeCoverage(c) = \sum_{c' \in CoverageSet(c)} \frac{1}{|ReachabilitySet(c')|} \tag{4.54}$$

[8]CNN is a method proposed to reduce the storage requirements of the original data set, for the efficient implementation of the nearest neighbor decision rule in pattern classification problems.

```
1. OrderedSet ← Rank by Relative Coverage 2. CaseBase and NewCase
3. EvaluatedSet ← ∅
4. Changes ← true
5. WHILE Changes DO
6.       Changes ← false
7.       FORALL Case IN OrderedSet DO
8.            IF EvaluatedSet can not solve Case THEN
9.               Changes ← true
10.              Add Case to EvaluatedSet
11.              Remove Case from OrderedSet
12.           ENDIF
13.      ENDFOR
14. ENDWHILE
15. RETURN EvaluatedSet
```

Figure 4.46: The relative coverage criteria algorithm.

Our implementation of this criteria is detailed in figure 4.46. $CaseBase$ and $NewCase$ are respectively the set of cases in the case base and a new case not yet in the case base.

The algorithm starts by ranking by relative coverage the cases in the case base to the *OrderedSet* list, including the new case. It sets the list of evaluated cases (*EvaluatedSet*) to empty and the *Changes* flag to true. While there are changes in the list of evaluated cases, determined by the *Changes* flag, then: turn *Changes* to false; for each case in *OrderedSet*, if cases in the evaluated list can not solve it, then set *Changes* to true, add the current case to the evaluated list and remove the case from *OrderedSet*. At the end return the evaluated list.

Using this algorithm if the $NewCase$ makes part of the $EvaluatedSet$ then it should be added to the case base otherwise it should be suggested for deletion. Cases in the case base that are not in $EvaluatedSet$ should be removed from the case base.

4.6.8 Relative Performance Metric Criteria

Leake and Wilson [Leake and Wilson, 2000] developed this criteria based on the notion of relative performance to decide if a case should be added to the case base or not. The relative performance of a case is defined as follows:

$$RP(c) = \sum_{c' \in CoverageSet(c)} \left(1 - \frac{AdaptCost(c, c')}{\max_{c'' \in ReachabilitySet(c') - \{c\}} \{AdaptCost(c'', c')\}} \right) \quad (4.55)$$

where *AdaptCost* is the adaptation cost of transforming c into c'.

In REBUILDER, *AdaptCost* is defined as the modulus of the difference between objects in c and objects in c'. A submitted case should be added to the case base if it's

relative performance is higher than a threshold value defined by the KB administrator. In our experiments we have used 0.5 as the threshold value.

4.6.9 Competence-Guided Criteria

The competence-guided criteria [McKenna and Smyth, 2000] extends previous works of Smyth, and uses the notions of case competence based on case coverage and reachability. This criteria uses three ordering functions:

Reach for Cover (RFC): uses the size of the reachability set of a case. The RFC evaluation function implements this idea: the usefulness of a case is an inverse function of its reachability set size.

Maximal Cover (MCOV): is based on the size of the coverage set of a case. Cases with large coverage sets can classify many target cases and in this way must make a significant contribution to classification competence.

Relative Coverage (RC): is defined in the relative coverage criteria (see subsection 4.6.7).

In REBUILDER the algorithm that implements this criteria is presented in figure 4.47 ($CaseBase$ and $NewCase$ are defined as in the relative coverage and condensed NN criteria). This algorithm starts by initializing the set of remaining cases (*RemainingSet*) with all the cases in the case base and the new case being evaluated. Then the edited set of cases (*EditedSet*) is initialized to empty. While there are cases in *RemainingSet*, the algorithm gets the case (*Case*) in *RemainingSet* according to the selected ordering function, adds it to the *EditedSet*, removes all cases of the *Case*'s coverage set from *RemainingSet*, and updates the reachability and coverage sets of the cases in the *RemainingSet*. The $NewCase$ is only added to the case base if it is part of the *EditedSet*.

1. $RemainingSet \leftarrow CaseBase \cup NewCase$
2. $EditedSet \leftarrow \varnothing$
3. WHILE $RemainingSet \neq \varnothing$ DO
4. $\quad Case \leftarrow$ Next case in the $RemainingSet$ according to the selected ordering function
5. $\quad$ Add $Case$ to $EditedSet$
6. $\quad$ Remove all cases in $CoverageSet(Case)$ from $RemainingSet$
7. $\quad$ Update the reachability and coverage sets of the cases in the $RemainingSet$
8. ENDWHILE
9. RETURN $EditedSet$

Figure 4.47: The competence-guided criteria deletion algorithm.

4.7 Word Sense Disambiguation

In REBUILDER objects must have an associated synset, otherwise the reasoning mechanisms are not able to work properly. REBUILDER uses word sense disambiguation (WSD) mechanisms to determine which is the correct synset for an object. To obtain the correct synset for an object, REBUILDER uses the object's name, the other objects in the same class diagram, and the object's attributes if it is a class.

The object's class diagram defines the context in which an object is referenced, so it is used to disambiguate the meaning of the object. To illustrate this, suppose that an object named *board* is created. This object can mean, either a piece of lumber, or a group of people assembled for some purpose. This name has two possible synsets, one for each meaning. But suppose that there are other objects in the same diagram, such as *board member* and *company*. These objects can be used to select the correct synset for board, which is the one corresponding to "group of people".

The disambiguation process starts by extracting from WordNet the synsets corresponding to the object's name. This requires the system to parse the object's name, which most of the times corresponds to a composition of words. REBUILDER uses specific heuristics to choose the words to be used. For instance, only words corresponding to nouns are selected, because usually objects correspond to entities. A morphological analysis must also be performed, extracting the regular noun from the word. After this, a word or a composition of words has been identified and will be used to search the WordNet. The result of this search is a set of synsets. From this set, REBUILDER can use several WSD methods to select one synset.

There are several WSD methods, to assess which one works better with our approach, we implemented and tested several methods. We selected 9 methods (most

of them described in [Rigau et al., 1997]) and added two new methods, which are described in the next subsections. For the other methods, we also show how they were defined in REBUILDER. Most of them integrate some changes concerning to the original definition. We have made a change to all methods to improve their performance, which consists on returning the synset selected by the Entry Sense Ordering method (described ahead) whenever a method can not chose a synset.

4.7.1 Monosemous Genus Term

If the target word has only one candidate synset, then this method selects it, otherwise this method fails. This method is used in combination with other methods. At the beginning of the application of each method we use this one to test if the target word is monosemous, if it is the word's synset is found. WordNet 1.7.1 has 111276 words, from which 82% are monosemous, if we consider only noun synsets. Despite these figures, the most frequent used words are not monosemous.

4.7.2 Entry Sense Ordering

This method is based on the sense ordering of WordNet, which uses the occurrence frequency of concepts in the Brown Corpus [Francis and Kuçera, 1982].

REBUILDER uses the synsets of WordNet ordered by occurrence frequency. When a word needs to be disambiguated this method selects the most frequent synset.

4.7.3 Word Matching

Associated with each WordNet synset there is a glossary entry, which is used by this method to find the correct synset for the target word. The candidate synsets are ranked using the following score:

$$WMScore(w,s) = \#\Big[\big[\{w\} \cup Context(w)\big] \cap Glossary(s)\Big] \tag{4.56}$$

where w is the target word, s is a candidate synset, $Context(w)$ is a function that returns the list of context words associated with w. $Glossary(s)$ returns the list of words in the s definition. This method intersects the target word and the context

words with the words in the glossary entry of each synset, selecting the one with more words in common. We only consider nouns, the other word categories are ignored.

4.7.4 Simple Cooccurrence

This method uses cooccurrence data derived from WordNet glossary. Two words co-occur if they appear in the same definition. For each word it is collected a list of cooccurrence frequencies with other words, based on the WordNet glossaries. Only frequencies different from zero are used. The cooccurrence frequency is defined by the number of times the two words appear in the same glossary divided by the number of times that the word being investigated appears in all the glossaries. It ranks candidate synsets using the score given by the formula:

$$SCScore(w,s) = \sum_{w_i \in Context(w) \wedge w_j \in Glossary(s)} cw(w_i, w_j) \tag{4.57}$$

where cw is the cooccurrence weight between two words given by cooccurrence frequency.

4.7.5 Cooccurrence Vectors

The cooccurrence vectors method takes the same data used in the previous method, but in a different way. The candidate synsets are ranked using the following score (function terms used here are the same as the ones used in the literature):

$$CVScore(w,s) = Sim(V_{Context(w)}, V_{Glossary(s)}) \tag{4.58}$$

where Sim is the vector multiplication operation. Sim's arguments are vectors. V represents the context of the words presented in the respective definitions:

$$V_{Definition} = \sum_{w_i \in Definition} Civ(w_i) \tag{4.59}$$

Civ is the cooccurrence information (cooccurrence frequency) vector of the words in the definition.

4.7.6 Semantic Vectors

WordNet *is-a* semantic relations define several tree-like structures, each one with one distinct root synset. There are nine noun roots used by this method to categorize synsets. It finds which are the roots of a specific synset going up in the *is-a* structure (a synset can have several parents, so a synset can have more than one semantic category). The ranking of the candidate synsets is given by:

$$SVScore(w, s) = Sim(V_{Context(w)}, V_{Glossary(s)}) \tag{4.60}$$

where Sim is the vector multiplication operation. V represents the context of the words presented in the respective definitions:

$$V_{Definition} = \sum_{w_i \in Definition} Swv(w_i) \tag{4.61}$$

Swv is the salient word vector of the words in the definition. This vector is based on the top sense categories for nouns of WordNet[9], and its elements comprise ones and zeros. One if the category is present, zero if not.

4.7.7 Conceptual Distance v1.0

This method was the first one we developed, and is based on the distance between synsets in the WordNet structure. REBUILDER uses *is-a*, *part-of*, *member-of*, and *substance-of* relations to compute the distance between two synsets. Each candidate synset is given a score based on the formula:

$$CD1Score(w, s) = \sum_{s_i \in ContextSynsets(w)} WNDist(s_i, s) \tag{4.62}$$

where $ContextSynsets(w)$ returns the list of synsets for each context word (one synset per context word), which implies that context words must result from a process of previous disambiguation. This can be performed in various ways. In our experiments we use a combination of methods. $WNDist$ returns the WordNet distance between two synsets and is given by:

$$WNDist(s_1, s_2) = 1 - \frac{1}{\ln(SDist(s_1, s_2) + 1) + 1} \tag{4.63}$$

[9]The top categories in version 1.7.1 are: physical thing, psychological feature, abstraction, state, event, human activity, group, possession, and phenomenon.

where $SDist$ (semantic distance) returns the number of links between synsets s_1 and s_2.

4.7.8 Conceptual Distance v2.0

The difference of this version to the previous one, is that this works with context words not yet disambiguated. Version 2 of the conceptual distance uses all the synsets of all context words to compute a score for each candidate score. The formula used for ranking candidate synsets is:

$$CD2Score(w, s) = \sum_{w_i \in Context(w)} ShortestDist(w_i, s) \tag{4.64}$$

where $ShortestDist$ is the minimum distance between s and all synsets of w_i. The definition is:

$$ShortestDist(w_i, s) = \min\{s_i \in Synsets(w_i) : WNDist(s_i, s)\} \tag{4.65}$$

$Synsets(w)$ returns the synsets associated with word w.

4.7.9 Information Content

The information content method was developed by Resnik [Resnik, 1995b] and is based on the measurement of the semantic similarity in an *is-a* taxonomy. It uses the concept of information content of each hierarchy's node, which is assessed using the frequency of words in a text corpus. As Resnik, we use the Brown Corpus of American English [Francis and Kuçera, 1982] to extract the noun frequencies. Candidate synsets are ranked using the following formula:

$$ICScore(w, s) = \sum_{w_i \in Context(w)} IC(w_i, s) \tag{4.66}$$

where IC yields the information content of a word, defined by:

$$ICScore(w_i, s) = \max\{s_i \in Synsets(w_i) : RDist(MSCA(s_i, s))\} \tag{4.67}$$

$MSCA$ is the synset corresponding to the Most Specific Common Abstraction between two synsets, and the $RDist$ is the Resnik distance and is given by:

$$RDist(s) = -\frac{\ln(\sum_{s_i isSubsumedBy(s)} CProb(s_i))}{k} \tag{4.68}$$

The synsets (s_i) are the WordNet synsets subsumed by s. With $k = 3$ we obtained the best results. $CProb$ is the concept probability and is given by:

$$CPRob(s) = \sum_{w_i \in Words(s)} WordFrequency(w_i) \tag{4.69}$$

where $Words(s)$ returns the words in WordNet associated with synset s.

4.7.10 Lists of Words

Nastase and Szpakowicz [Nastase and Szpakowicz, 2001] developed a method for WSD using the words of synsets related to the candidate synsets.

In our implementation of the list of words method we use the candidate synset words and the words of the neighbors synsets of the candidate synset. Neighbor is any synset at one relation distance from candidate synset (in the formula we distinguish from *is-a* neighbors and any other type of relations, which can be *member-of*, *substance-of* or *part-of*). The candidate synsets are ranked using the following formula:

$$\begin{aligned} LWScore(w, s) = [&10^6 \cdot WordIntersection(w, s) + \\ &\sum_{s_i \in Fathers(s)} (10^4 \cdot WordIntersection(w, s_i)) + \\ &\sum_{s_i \in Children(s)} (10^2 \cdot WordIntersection(w, s_i)) + \\ &\sum_{s_i \in Neighbors(s)} WordIntersection(w, s_i)] \cdot 10^{-7} \end{aligned} \tag{4.70}$$

The function $WordIntersection(s)$ performs the intersection of $Words(s)$ with *ContextWords(w)*, $Fathers$ returns the synset's fathers using the *is-a* links, $Children$ yields the children using the *is-a* relations, and $Neighbors$ the nodes adjacent to s using the other types of relations. The equation 4.7.10 is equivalent to ranking candidate synsets first by intersection with context words, then by fathers, children and neighbors. We use a numerical formula because it is more efficient and we needed a score to be used in the next method (Method Combination).

4.7.11 Method Combination

We have combined all the methods described before in an unique method, by taking each method contribution in the selection of the best synset. With the exception of Entry Sense Ordering, the contribution of all the other methods is the score of the best candidate synset. In the Entry Sense Ordering this score is:

$$\frac{1}{NumberOfCandidateSynsets} \tag{4.71}$$

Each method attributes a score to one synset (the best synset according to that method), and each candidate synset sums up its contributions. The one with the higher score is chosen as the best synset. The idea of combining several methods is not new (see [Rigau et al., 1997]), what is new in this approach is the form of combination.

Chapter 5

Experimental Evaluation

This chapter presents the experimental work performed with REBUILDER. The goal is to make an assessment of our approach and to select the configuration parameters to be used in various reasoning modules of REBUILDER.

The evaluation comprises tests at two levels: reasoning mechanisms and field tests. The first part was accomplished and is fully described in this chapter. The field testing has not been performed due to the dimension and characteristics of REBUILDER. The problem of field testing is that it should be performed in a real development environment in order to be accurate. We faced a main difficulty concerning this type of tests at this phase of the project, which is, in order for REBUILDER to be fully tested, it should be integrated into a software developing company, or team, which has not been possible yet. This needs time and commitment at the higher levels of the company, because the changes needed in the hosting organization are considerable, and the development process would have to be adapted to the REBUILDER's framework for software development. So that all the potentialities of the platform can be used. In this way this phase will need more time and a strong involvement of a software house. Another issue that arises is that in order to have the field testing done, we need a more stable version of REBUILDER. This can only happen with a preliminary testing of the reasoning modules and of their interactions. In this way, we have planned and executed the preliminary testing. The next, we will perform the field tests, which are out of the scope of this thesis.

The preliminary experiments that we performed are presented in this chapter accordingly to the chronological sequence in which the modules were developed. In

Table 5.1: The outline of this chapter, with the description of all performed experiments.

Tested Module	Experiment Description	Section
Retrieval	1 - Retrieval of partial cases.	5.2.1
	2 - Retrieval of partial classes.	5.2.2
	3 - Influence of indexing in retrieval.	5.2.3
	4 - Recall and precision.	5.2.4
Analogy	1 - Mapping algorithm and threshold value.	5.3.1
	2 - Combination of retrieval and analogy.	5.3.2
	3 - Importance of retrieval strategies for creative analogy.	5.3.3
Design Composition	1 - Design composition.	5.4
Design Patterns	1 - Design patterns.	5.5
Verification	1 - Effect of verification in analogy.	5.6.1
	2 - Effect of verification in design composition.	5.6.2
Learning	1 - Computational time efficiency.	5.7.1
	2 - Case base competence.	5.7.2
WSD	1 - First experiments with word sense disambiguation.	5.8.1
	2 - Comparison of WSD methods in REBUILDER.	5.8.2

short, the reasoning modules development had the following sequence: retrieval, analogy, design composition, design patterns, verification and learning. The WSD experiments and developments were performed in two steps, but in parallel with other modules. We follow this sequence for presentation of the tests in this chapter. Table 5.1 provides an outline for this chapter.

Before presenting the tests on REBUILDER we describe the knowledge base used for the experiments, then each section of this chapter introduces an experiment, which most of the times concerns to a specific function of the system.

It is important to stress that most of our experiments can not be compared against other systems, because there is no system with enough similarity to REBUILDER to make comparative tests. Related systems, which are described in chapter 6, do not use UML or even work in the same way as our system, in terms of input and output data.

5.1 Experimental Knowledge Base

The Knowledge Base used in the experiments comprises a case library, WordNet, case indexes, data type taxonomy, and a set of query problems. We used WordNet version

1.7.1, and we considered the noun synsets (about 78000) and semantic relations (in the number of 97000).

The case library comprises 60 software design cases. These cases were collected mostly from academic projects and examples from text books. They come from four different domains: banking information systems, health information systems, educational institution information systems, and store information systems (grocery stores, video stores, and others). Each design comprises a package, with 5 to 20 classes (the total number of classes in the knowledge base is 586). Each object has up to 20 attributes, and up to 20 methods. The designs are defined at a conceptual level, so the design is at an early stage of development and has only the fundamental classes, attributes and methods.

5.2 Retrieval Experiments

This section describes the experiments performed with the retrieval module. The first experiments were performed to assess the package retrieval (or case retrieval) performance. Three problem sets were used, all generated from the case library. For each problem, we defined a set of relevant cases, with the best case also identified. These experiments were used for exploratory purposes, helping to identify flaws and possible improvements in the retrieval mechanism. After this starting experiment, we performed the same process for class retrieval.

At this point our intuition was that misclassification of software objects had a strong role on the accuracy of the retrieval mechanism. So, some experiments were performed to assess the degree of influence of misclassification or wrong indexing in retrieval.

These three experiments were the first ones to be performed on the retrieval mechanism, and they gave several insights to improve the retrieval process. We then made several modifications, the most important ones being the two new versions of the retrieval mechanism that were incorporated into REBUILDER and several improvements in the similarity metric. After these changes, more experiments were performed. In these experiments we used a set of different target problems that were not generated from the cases in the case base. We tried to assess the retrieval

Table 5.2: The configurations weights for the package retrieval experiments.

	ω_1	ω_2	ω_3	ω_4
Configuration 1 (C1)	0	1	0	0
Configuration 2 (C2)	0	0.75	0.25	0
Configuration 3 (C3)	0	0.5	0.5	0
Configuration 4 (C4)	0	0.25	0.75	0
Configuration 5 (C5)	0	0	1	0

mechanism performance in terms of recall and precision. We also evaluated the three versions of the retrieval algorithms, and performed experiments concerning weights for the package similarity metric, and the size of the retrieval set.

5.2.1 Retrieval of Partial Cases

The goal of these experiments was to provide a preliminary assessment of the retrieval mechanism. Three sets of package problems - P20, P50 and P80 - were specified, based on the case library. P20 is a set of 25 incomplete problems. Each problem is a case copy with 80% of its objects deleted. Attributes and methods are also reduced by 80%. Sets P50 and P80 have the same problems, but with, respectively, 50% and 20% of their components deleted. One of the experiment goals is to define the best weight configuration for package retrieval.

For each problem, a best case and a set of relevant cases were defined previously to the run of the experiments. These sets were used to evaluate the accuracy of the retrieval algorithm. For each set of problems, the weight configurations (for formula 4.1) that were used are presented in table 5.2. Because each case has only one package, weights ω_1 and ω_4 are not used, leaving only weights ω_2 (diagram similarity) and ω_3 (categorization similarity), which concern to the class diagram similarity and package type similarity. Configuration C1 assesses only the diagram similarity, while C5 addresses only the categorization similarity, the other configurations are in the middle of the spectrum combining diagram similarity with type similarity. For each problem run, we analyzed the 20 cases with the highest retrieval score. The gathered data were: best case is first (yes or no), best case is selected (yes or no), percentage of the relevant cases retrieved and the best 20 cases ranked by similarity.

From the results presented in table 5.3, it can be inferred that configuration C4

Table 5.3: Results for package retrieval obtained in terms of percentage: relevant cases retrieved, best cases in the first ranking place, and best case retrieved by number of cases retrieved.

	C1	C2	C3	C4	C5
Relevant Cases Retrieved	82.03	84.15	84.75	85.12	83.95
Best Case First	81.33	85.33	88.00	88.00	61.33
Best Case Retrieved (in 5 cases)	88.00	92.00	92.00	92.00	85.33
Best Case Retrieved (in 10 cases)	89.33	93.33	93.33	93.33	93.33
Best Case Retrieved (in 15 cases)	92.00	93.33	93.33	93.33	93.33
Best Case Retrieved (in 20 cases)	93.33	93.33	93.33	93.33	93.33

has the higher percentage of relevant case retrieval, though results obtained for C2, C3 and C5 are very close to the ones obtained with C4. In terms of best case retrieval (among the first 20 relevant cases) all the configurations retrieved the best case in 93.3% of the runs, though results for the retrieval of the best case in the first ranking position are distinct, with C3 and C4 achieving the best results. In terms of time performance there are no significant differences along the five configurations.

Figure 5.1 shows the average retrieval of relevant cases by retrieval set size and configuration used. It can be seen that configuration C5 has the best performance followed closely by C4 and C3. C3, C4 and C5 reach the same value for twenty cases. Configuration C2 has a lower performance and does not reach the same final value as the former configurations. Finally C1 has the lowest performance.

The results for the relevant cases retrieved by configuration and by problem set are presented in figure 5.2. Configuration C1 achieves the worst results, and from the other results it is perceived a tendency for type similarity to be more important than diagram similarity, although this is not a straightforward conclusion. For instance, with the increase in the target problem size, the best result shifts from C5 to C4, and then to C4 and C3.

The results presented in figure 5.3 concern the best case retrieval and what is observed is that problem set P20 achieves the best results, independently of the configuration that is used. An explanation for this, is that the configuration similarity does not influence retrieval. It only influences ranking of cases. Since these results concern a retrieval set of P20 problems, the retrieval algorithm is able to retrieve the best case independently of the similarity configuration. If we look at the results for the retrieval set of size one (see figure 5.4) this observation does not hold. Configuration

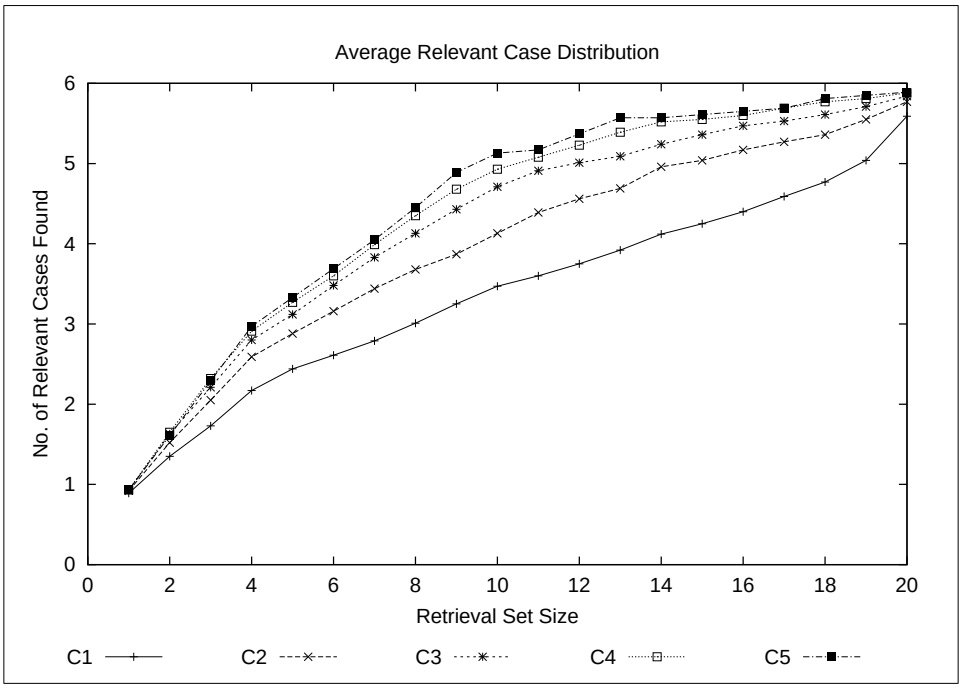
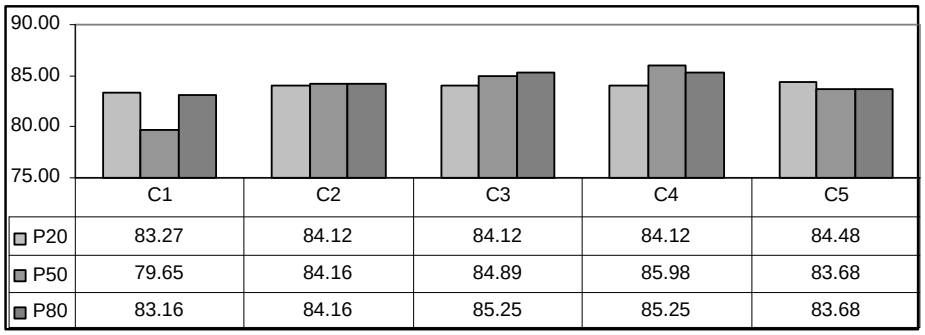
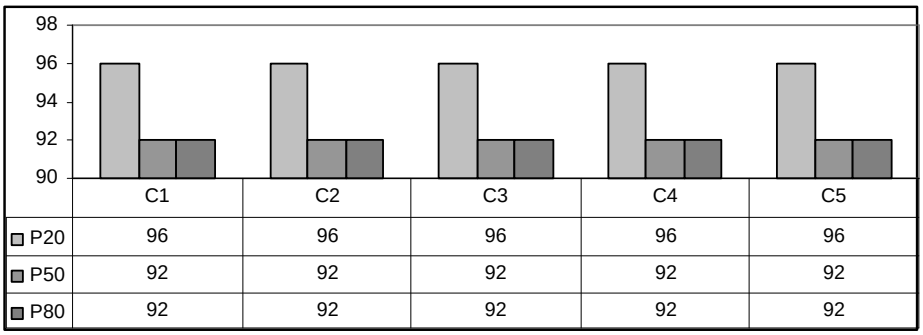


Figure 5.1: The results obtained for the average relevant cases found by configuration and retrieval set size.



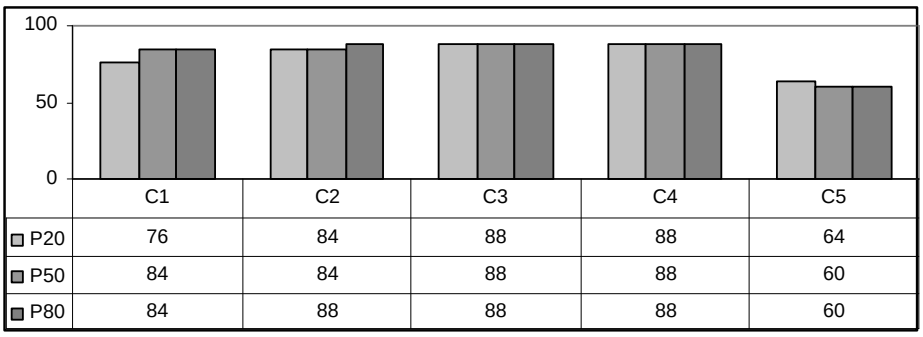
	C1	C2	C3	C4	C5
P20	83.27	84.12	84.12	84.12	84.48
P50	79.65	84.16	84.89	85.98	83.68
P80	83.16	84.16	85.25	85.25	83.68

Figure 5.2: The complete results for the relevant cases retrieved by configuration and problem set (results in percentage).



	C1	C2	C3	C4	C5
P20	96	96	96	96	96
P50	92	92	92	92	92
P80	92	92	92	92	92

Figure 5.3: The complete results for the best case retrieved by configuration and problem set (results in percentage).



	C1	C2	C3	C4	C5
P20	76	84	88	88	64
P50	84	84	88	88	60
P80	84	88	88	88	60

Figure 5.4: The complete results for the best case first by configuration and problem set (results in percentage).

C5 is clearly the worst because it does not take into account diagram similarity, and C3 and C4 are the best configurations due to the compromise they achieve between type and diagram similarity, though it can be seen a tendency for type similarity to have a stronger importance than diagram similarity.

In conclusion it can be seen that configuration C4 achieves the best overall results, while C1 achieves the worst results. In relation to the problem size, the trend that is observed in the results is that, longer queries achieve better retrieval accuracy if diagram similarity has a higher value. But this is only an observed tendency, more experiments and a bigger case base are necessary to support this observation.

5.2.2 Retrieval of Partial Classes

For the class retrieval experiments we used the same idea of the previous subsection. Three sets of class problems were specified based on cases. The strategy used for

Table 5.4: The configurations weights for the class retrieval experiments.

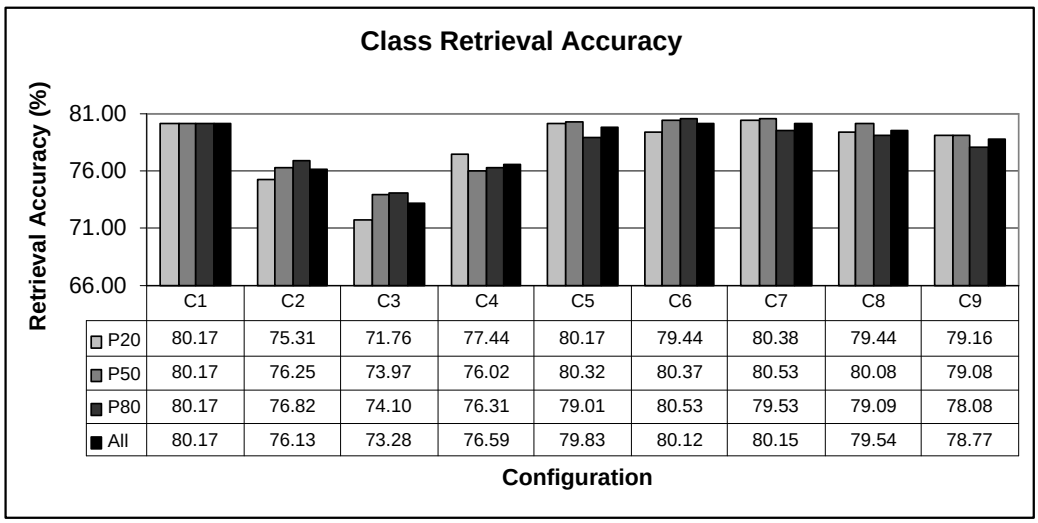
	ω_1	ω_2	ω_3
Configuration 1 (C1)	1	0	0
Configuration 2 (C2)	0.5	0.5	0
Configuration 3 (C3)	0	1	0
Configuration 4 (C4)	0.5	0.25	0.25
Configuration 5 (C5)	0.25	0.5	0.25
Configuration 6 (C6)	0.5	0	0.5
Configuration 7 (C7)	0.25	0.25	0.5
Configuration 8 (C8)	0	0.5	0.5
Configuration 9 (C9)	0	0	1

problem definition is the same as the one for problem packages (see subsection 5.2.1). An incomplete set P20, with 25 problems, being each problem a copy of a class from a case in the library with 80% of its methods and attributes deleted. The other sets have the same problems but with 50% (P50) and 20% (P80) of the methods and attributes deleted. For each class, a set of relevant classes was defined before the running of the experiments. These sets were used to evaluate the accuracy of the algorithm.

For each problem set, the weight configurations (for formula 4.6) that were used are presented in table 5.4. Weight ω_1 concerns the class categorization similarity, ω_2 the inter-class similarity, and ω_3 the intra-class similarity (see subsection 4.1.2). For each run the best 20 retrieved classes were analyzed. The data gathered is: percentage of the relevant classes retrieved, and the best 20 classes ranked by similarity.

The accuracy results for class retrieval are presented in figure 5.5 and table 5.5. Configurations C1, C6 and C7 have the best results, though the difference to C5 and C8 is small. These results show a tendency to obtain the higher accuracy values with emphasis on the type similarity, and then, in a lower level of importance, with the increase in the weight for the inter-class similarity. Figure 5.6 presents the average distribution for class retrieval by configuration and size of the retrieval set. As can be seen, configuration C3 has the worst performance. The ranking concerning performance, going from better to worst is: C1 - C5, C6 and C7 - C8 - C2 and C9 - C4 - C3.

As the experimental results show, in class retrieval the categorization similarity



	C1	C2	C3	C4	C5	C6	C7	C8	C9
P20	80.17	75.31	71.76	77.44	80.17	79.44	80.38	79.44	79.16
P50	80.17	76.25	73.97	76.02	80.32	80.37	80.53	80.08	79.08
P80	80.17	76.82	74.10	76.31	79.01	80.53	79.53	79.09	78.08
All	80.17	76.13	73.28	76.59	79.83	80.12	80.15	79.54	78.77

Figure 5.5: Results obtained for class retrieval by configuration and problem set.

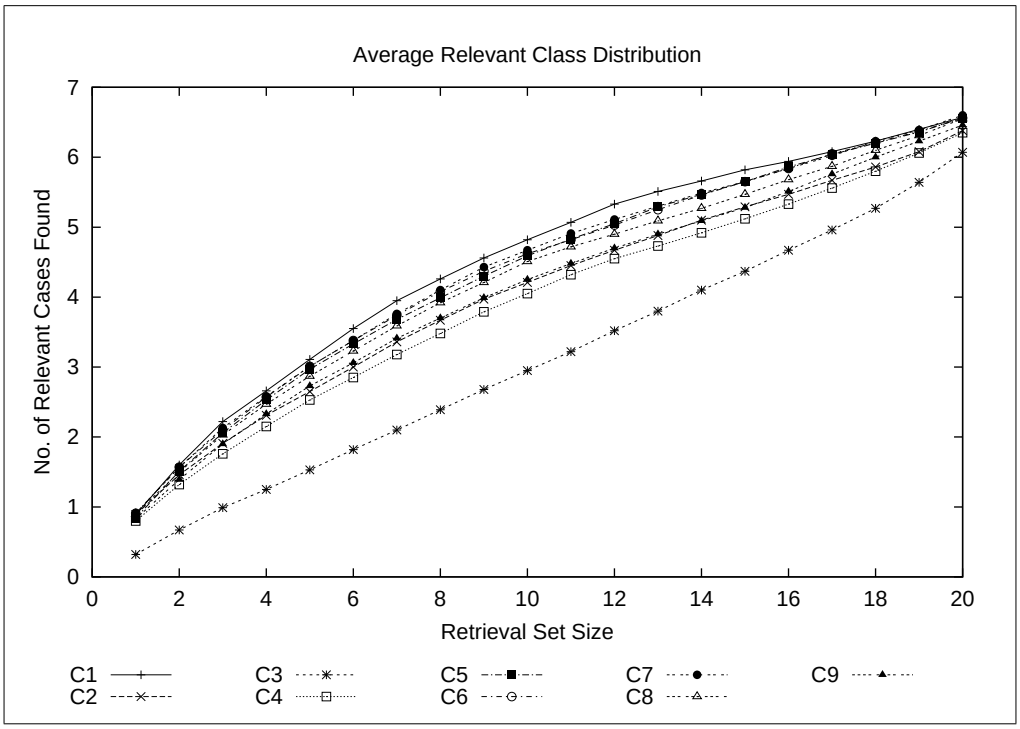


Figure 5.6: The results obtained for the average relevant classes found by configuration and size of retrieval set.

Table 5.5: The accuracy results obtained for class retrieval.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
Retrieval Accuracy	80.17	76.13	73.28	76.59	79.83	80.12	80.15	79.54	78.77

is determinant for finding the right classes, which is consistent with knowledge on software development. The conclusion that we can get from these results is that the best configuration to be used, based on these results, is C1, C6 or C7.

5.2.3 Influence of Indexing in Retrieval

In this subsection we present results obtained with REBUILDER on the effect of object classification on the accuracy of package retrieval. We used two knowledge bases with exactly the same objects (KB 1.0 and KB 1.1, which is the same KB used for the previous experiments). The difference is that KB 1.0 has 16% of misclassified objects, while KB 1.1 does not have misclassifications. Misclassification is due to the polysemy phenomenon in WordNet (see section 3.3.2). We tested the package retrieval algorithm with three sets of package problems based on the library of cases (P20, P50 and P80 defined in subsection 5.2.1).

For each problem we previously defined a best case (the case from which we created the problem), and a set of relevant cases with no ranking. For each running cycle we retrieved the best 20 cases. Three retrieval characteristics were analyzed: percentage of times that the best case was selected as the first in the retrieval ranking (first column in table 5.6); percentage of relevant cases retrieved (second column in table 5.6); and percentage of best case retrieval in any position (third column in table 5.6). The results show that 16% of object misclassification can lower the retrieval accuracy by 31.73%, which is a significant result.

5.2.4 Recall and Precision

Experiments presented in this subsection have the goal of determining the properties of each of the three versions of the retrieval algorithm (see subsection 4.1.1). It is important to remind that the algorithm versions differ only on the initial set of synsets, which corresponds to start the WordNet search at different places. Another

Table 5.6: Results obtained when comparing the influence of object misclassification in retrieval accuracy.

	Best Case First	Relevant Cases Retrieved	Best Case Retrieved
KB 1.0 - P20	56.00	58.64	68.00
KB 1.0 - P50	44.00	57.84	64.00
KB 1.0 - P80	40.00	57.84	64.00
KB 1.0 - All	46.67	58.11	65.33
KB 1.1 - P20	88.00	84.12	96.00
KB 1.1 - P50	88.00	85.98	92.00
KB 1.1 - P80	88.00	85.25	92.00
KB 1.1 - All	88.00	85.12	93.33

objective was to determine which is the best weight configuration for the package retrieval similarity metric. This is decisive in ranking the cases.

Twenty five problems were defined, each one having one package with several classes (between 3 and 5), which were related to each other by UML associations or generalizations. These problems are distributed by the four case domains in the following way: banking information systems (6 cases), health information systems (7 cases), educational institution information systems (3 cases), and store information systems (9 cases). For each problem we identified the relevant set of cases from the case library. These sets are used for comparison with the set of cases retrieved by REBUILDER, yielding the recall[1] and precision[2] of the retrieval and similarity mechanisms.

For each problem REBUILDER retrieved a set of cases. We tested for sets of different sizes in order to study recall and precision values along sets of different sizes. We defined three configurations corresponding to each retrieval version:

- ***C1*** - simple version.
- ***C2*** - extended version.
- ***C3*** - ambiguous version.

We tested these configurations on sets of: 1, 3, 5, 10, 15, and 20 cases. The values for the weights of formula 4.1 (the package similarity), are presented in table 5.7.

[1]The number of relevant cases retrieved divided by the total number of relevant cases in the case base.

[2]The number of relevant cases retrieved divided by the total number of retrieved cases.

Table 5.7: Weight configurations used for tests.

Configuration	Diagram Similarity	Type Similarity
W1	1	0
W2	0.75	0.25
W3	0.5	0.5
W4	0.25	0.75
W5	0	1

Table 5.8: F-Measure values obtained for weight configurations and retrieval algorithm configurations.

Configuration	W1	W2	W3	W4	W5
C1	0.515	0.537	0.520	0.520	0.435
C2	0.531	0.543	0.537	0.537	0.454
C3	0.530	0.552	0.535	0.535	0.435

Data collected on each run was the recall and precision for the retrieval configuration. Because recall and precision are antagonistic measures we used the F-Measure defined by [Rijsbergen, 1979]. The F-Measure combines recall and precision in a single measure as defined in equation 5.1. The F-Measure results, by algorithm configuration and weight configuration, are shown in table 5.8.

$$F = \frac{2 \cdot Recall \cdot Precision}{Recall + Precision} \tag{5.1}$$

These results show that the best answers are achieved with configurations *C3* and *W2*, which comprise the use of the ambiguous retrieval algorithm with the similarity weight on the diagram similarity higher than the weight on type similarity. *W2* configuration outperforms all the other configurations, which are similar to each other, except for configuration *W5* (based only on type similarity). From these results it can be inferred that the class diagram similarity is more important for case similarity than type similarity.

Concerning the algorithm comparison, with the exception of *W2*, all the other weight configurations indicate that *C2* is the algorithm that shows the best performance. These values are achieved with a retrieval set of size five, which yield the best results.

Table 5.9: F-Measure results obtained for the retrieval algorithm configurations varying the size of the retrieval set.

	1	3	5	10	15	20
C1	0.249	0.459	0.537	0.465	0.400	0.329
C2	0.249	0.479	0.543	0.478	0.415	0.327
C3	0.279	0.479	0.552	0.478	0.402	0.327

Table 5.10: Recall and precision values obtained for the retrieval algorithm configurations varying the size of the retrieval set.

	Recall			Precision		
	C1	C2	C3	C1	C2	C3
1	0.149	0.149	0.169	0.760	0.760	0.800
3	0.350	0.370	0.370	0.667	0.680	0.680
5	0.497	0.502	0.517	0.584	0.592	0.592
10	0.608	0.633	0.633	0.376	0.384	0.384
15	0.698	0.712	0.693	0.280	0.293	0.283
20	0.707	0.697	0.697	0.214	0.214	0.214

We performed an experiment using the weight configuration *W2*, in which we varied the size of the retrieval set. We also compared the algorithm configurations. As presented in table 5.9, the results confirm that the best size for the retrieval set is five. Configuration *C3* obtains the best results up to the size of ten, and after this *C2* outperforms the other configurations for sizes 10 and 15. For size 20 the best results are achieved by *C1*. These results show that the size of the retrieval influences the F-Measure for retrieved cases.

To provide an idea about the recall and precision values that were achieved, table 5.10 presents the recall and precision values for the F-Measure values presented in table 5.9. It can be pointed that while recall increases with the size of the retrieved set, precision decreases. The best compromise is found with the retrieval set of size five.

F-Measure values provide an idea of the quality of the cases that are retrieved and how they are being ranked. In order to assess performance we have also collected the retrieval and ranking times for each run, and we calculated the average computation time by algorithm version and weight configuration. The results achieved are described in table 5.11. They were calculated for a retrieval set of size five. The computational resources that were used were a PC Pentium III at 400 MHz with

Table 5.11: Average computation time values obtained for weight configurations and retrieval algorithm version, for each algorithm run (in seconds).

	W1	W2	W3	W4	W5
C1	22.117	21.877	22.838	22.335	0.332
C2	21.401	21.881	21.760	21.969	0.443
C3	21.074	21.134	21.856	21.148	0.329

Table 5.12: Average computation time for retrieval sets of different sizes by retrieval algorithm version (in seconds).

	1	3	5	10	15	20
C1	17.02	18.84	21.88	33.52	46.44	50.67
C2	13.57	19.18	21.88	39.07	50.30	54.88
C3	11.41	17.73	21.23	36.42	51.11	54.97

196 Mb of RAM running REBUILDER and a PC Pentium IV at 900 MHz with 384 Mb of RAM running the WordNet server. Computation times are very similar along all configurations with the exception of the weight configuration *W5* with the lowest computation time. This happens because this configuration does not compute the class diagram similarity (the weight value for the diagram similarity is zero), and the computation of diagram similarity takes about 98% of the class similarity computation time. Remember that diagram similarity computes structural similarity along with other demanding computations.

Scalability concerning computation time is shown in figure 5.7 and table 5.12. The results illustrate that the computation time by retrieved case (see table 5.13) decreases 75% for configuration *C3*, 80% for *C2*, and 85% for *C1*, which indicates that the method is scalable and that it shows a tendency to converge.

Table 5.13: Average computation time for one case, obtained for retrieval sets of different sizes by retrieval algorithm version (in seconds).

	1	3	5	10	15	20
C1	17.02	6.28	4.38	3.35	3.10	2.53
C2	13.57	6.39	4.38	3.91	3.35	2.74
C3	11.41	5.91	4.23	3.64	3.41	2.75

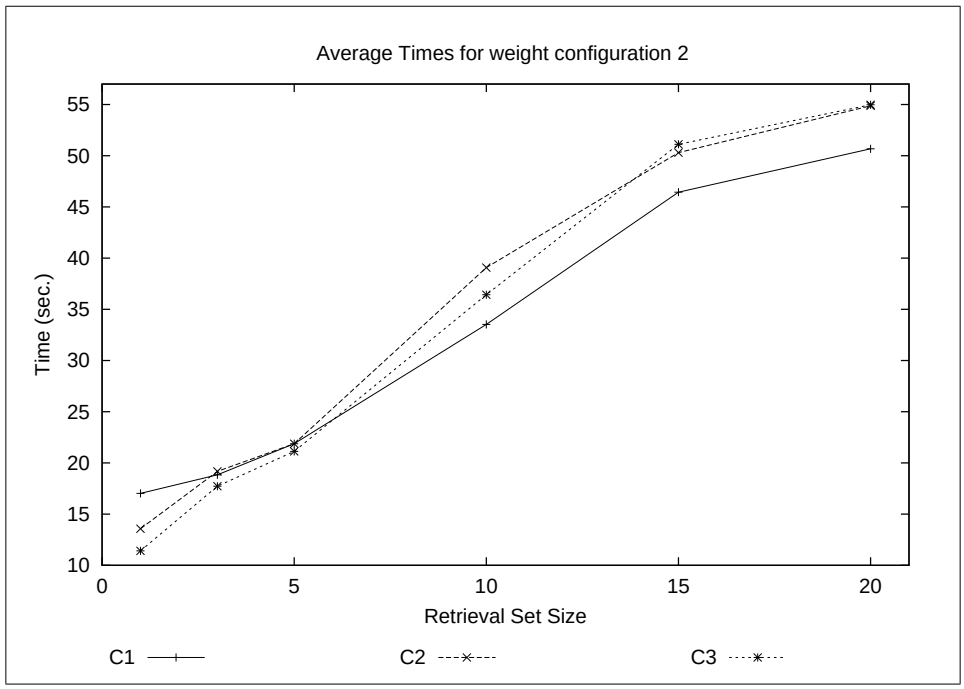


Figure 5.7: The computation time results obtained for different retrieval set sizes by retrieval algorithm version.

5.3 Analogy Experiments

This section describes the experimental tests developed to study the analogy module of REBUILDER. Three different experiments are presented. The first describes exploratory tests to determine the best mapping algorithm and threshold value. These are important experiments in order to establish good configuration parameters for the analogy mechanism. It is also important as a way to understand the analogy mechanism implemented and how it performs. The second set of tests analyze the use of retrieval for analogue candidate selection. The goal of these experiments is to define the best set of retrieval parameters to be used with the analogy module. This was performed before changes in the retrieval mechanism, yielding to three different retrieval strategies. The last experiment set comprises the three retrieval strategies in use.

Table 5.14: Configurations used in experiments.

Configuration	Mapping Algorithm	Threshold Value
C1	Relation-Based	0
C2	Relation-Based	0.25
C3	Relation-Based	0.5
C4	Relation-Based	0.75
C5	Relation-Based	1
C6	Object-Based	0
C7	Object-Based	0.25
C8	Object-Based	0.5
C9	Object-Based	0.75
C10	Object-Based	1

5.3.1 Mapping Algorithm and Threshold Value

The factor that constraints the analogy mapping is the value used as threshold for mapping objects and relations. This threshold is the minimum value allowed for mapping objects or relations (values near zero indicate that everything can be mapped). We also wanted to test the relation-guided and the object-guided mapping algorithms. Ten test configurations were prepared for experimentation (see table 5.14).

Each of the 60 cases in the case library was used as a problem, which yields 600 runs, 60 for each configuration since there are 10 different configurations. In each of these runs, we gathered the five best solutions for each mapping coming from different cases, according to ranking criteria (see section 4.2.2). This yields 20 solutions by run. The data gathered for each solution was: number of mappings, distance of mapped objects (in the WordNet), depth (in WordNet) of the $MSCA$ between mapped objects, and percentage of correct mappings by solution. This last figure was obtained by human evaluation. Several software developers evaluated the mappings. From the gathered data we derived the number of correct mappings by solution.

One of our hypotheses is that there is a correlation between correct mappings in analogical generated solutions and the distance of the mapped objects. Another hypothesi is that the probability of finding bizarre solutions, contrary to the probability of finding correct solutions, increases with the distance of mapped objects. Also we expected that the novelty of solutions presented by the analogy engine was also

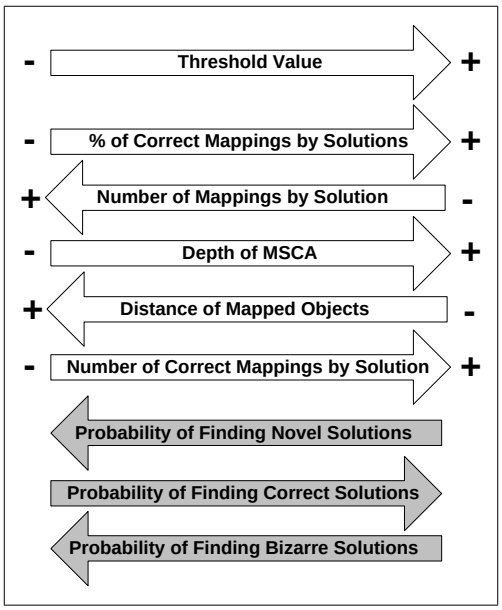


Figure 5.8: Hypotheses made for the experiments.

related to the distance of mapped objects and to the $MSCA$ depth. In this chain of thought, the threshold is the key, because it allows to vary the distance of mapped objects in the analogical engine. Figure 5.8 presents these hypotheses in a schematic form.

Results obtained for the average percentage of correct mappings are presented in the second column of tables 5.15 and 5.16. They show that with the increase in the threshold, the average percentage of correct mappings also increases. The object-based mapping algorithm outperforms the relation-based algorithm. Values for configurations C5 and C10 (threshold equal to 1) are zero because the algorithm does not find mappings. This happens because, for a mapping to have a similarity value of 1, the objects must have distance zero, the $MSCA$ must be equal to the objects and the depth of $MSCA$ must be the maximum depth of the WordNet *is-a* tree. While the first two values happen frequently, the last one is very unlikely to occur.

The other results presented in columns three to six of tables 5.15 and 5.16 support our hypotheses. While the semantic distance provides a way for measuring the similarity of the mapped concepts, the depth of $MSCA$ gives an insight on the level of abstraction at which the connection between the mapped concepts appears. We think that these two measures are related to novelty an unexpectedness, being novelty and

Table 5.15: Results obtained for the various analogy configurations using the relation-based mapping.

Configuration	Average % of Correct Mappings	Average Number of Mappings	Average Distance of Mapped Objects	Average Depth of $MSCA$	Average Number of Correct Mappings
C1	46.54	4.13	5.35	2.31	1.92
C2	46.12	4.13	5.42	2.28	1.91
C3	46.71	4.15	5.35	2.32	1.94
C4	64.67	3.83	3.37	3.25	2.48
C5	0.00	0.00	0.00	0.00	0.00

Table 5.16: Results obtained for the various analogy configurations using the object-based mapping .

Configuration	Average % of Correct Mappings	Average Number of Mappings	Average Distance of Mapped Objects	Average Depth of $MSCA$	Average Number of Correct Mappings
C6	54.35	4.20	4.97	2.51	2.29
C7	54.39	4.21	4.97	2.51	2.29
C8	54.66	4.16	4.91	2.52	2.28
C9	75.12	3.15	1.72	3.74	2.37
C10	0.00	0.00	0.00	0.00	0.00

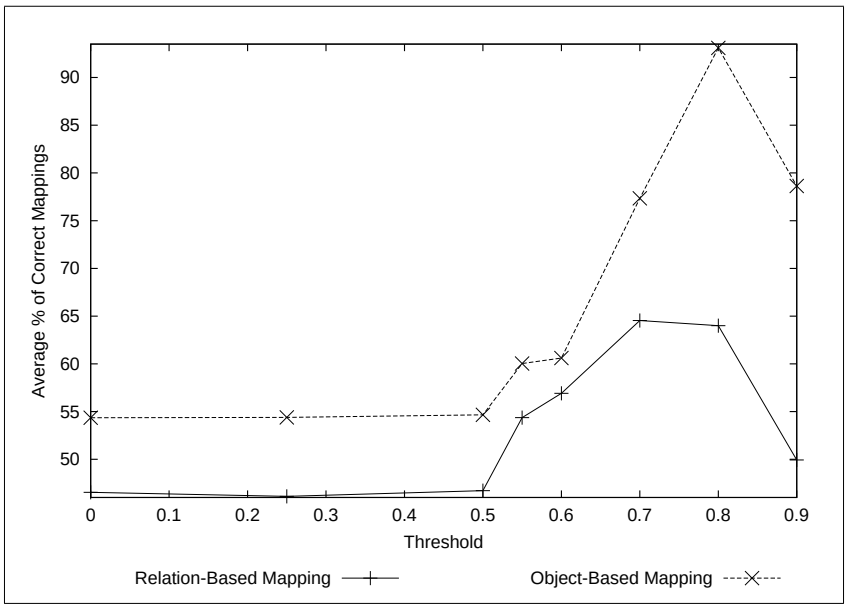


Figure 5.9: The results obtained for the average percentage of correct mappings by mapping algorithm and by threshold value.

unexpectedness directly proportional to the distance of mapped objects, and inversely proportional to the depth of the $MSCA$. As expected, experimental results support these hypotheses, showing that the probability of finding novel mappings increases when the threshold value for mapping objects decreases. But the possibility of finding bizarre mappings also increases, leading to a trade-off between the generation of novel solutions and correct solutions. The verification module can have an important role on filtering bizarre solutions.

In most situations the object-based algorithm presents better results than the relation-based approach (see tables 5.15 and 5.16). Figure 5.9 presents the average percentage of correct mappings for the mapping algorithms by threshold value. The object-based algorithm presents better results than the relation-based one. The object-based approach provides better results with a threshold of 0.8, and the relation-based approach with values 0.7 and 0.8. From these results we selected as default threshold 0.8.

The results comparing the mapping ranking criteria used in analogy and the threshold value for the average percentage of correct mappings are presented in figure 5.10. These results show that the best ranking criteria is the one that uses the number of mappings. The worst result is the one that uses the independence measure. The

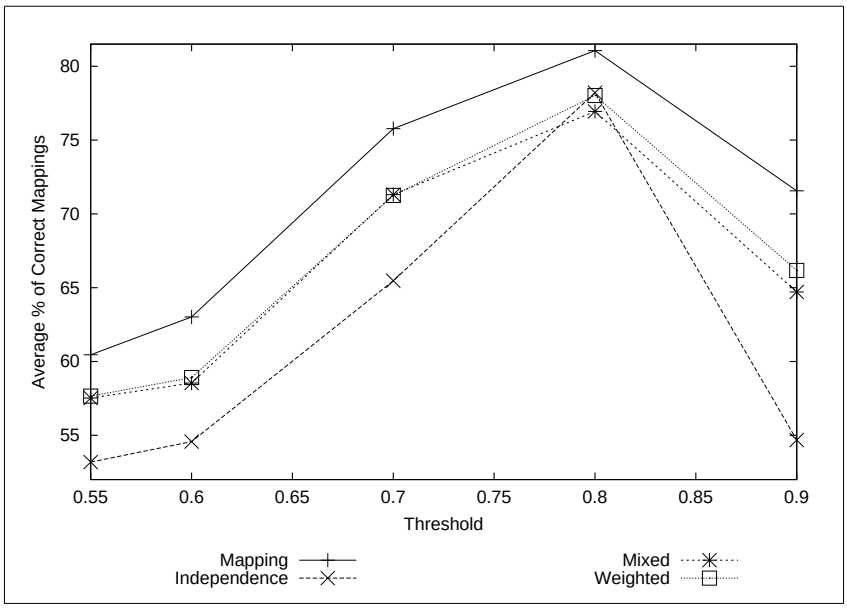


Figure 5.10: The results obtained for the average percentage of correct mappings by mapping ranking criteria and by threshold value.

Table 5.17: Run description for experiments involving the integration of CBR with Analogy.

Run	Similarity Type Used
C1 - Analogy Mapping	Structural
C2 - CBR + Analogy	Structural(0.5) + Semantic(0.5)
C3 - CBR + Analogy	Structural(1) + Semantic(0)
C4 - CBR + Analogy	Structural(0) + Semantic(1)

other two ranking criteria have similar performance, with the weighted criteria being slightly better. Once more, the best threshold value for all criteria was 0.8.

5.3.2 Combination of Retrieval and Analogy

These were the first experiments about the influence of candidate selection on analogy. We used the problem sets P20, P50 and P80. The goal of the experiments was to compare the candidate selection criteria. For each problem we applied the analogy mapping to all cases of the KB, thus serving as a base line for evaluation. The other candidate strategies are: CBR + Analogy using type similarity, CBR + Analogy using structural similarity, and CBR + Analogy using both type similarity and structural similarity. For each problem four runs were prepared, they are described in table 5.17.

For each test we analyzed the best five new cases, based on the mappings performed by the analogy algorithm. The collected data is: percentage of mappings in the new case considered correct, number of mappings, distance of mapped objects, depth of the Most Specific Common Abstraction ($MSCA$) between mapped objects, and computation time.

The results obtained for the first five solutions generated by the analogy module are presented in figure 5.11. The values presented are the average results of the three problem sets (P20, P50 and P80). These results show a clear trade-off between computation time and quality of results. Analogy by its own generates better solutions, which is made clear from the percentage of correct mappings. This is also reflected in the shorter semantic distance between objects, and the greater depth of the $MSCA$. Configurations which use case retrieval are faster than C1.

If we only analyze the results for the first generated solution (see figure 5.12), we conclude that the trade-off difference is smaller. For instance, the computation time for the first solution (the best one) is almost the same between C1 and C2, and the difference in the percentage of correct mappings is also small (5.13%). Probably the best alternative is to choose configuration C3, which loses only 5.52% of accuracy and is about 300 milliseconds faster (these are average values for the 75 problems).

5.3.3 Importance of Retrieval Strategies in Creative Analogy

The aim of these experiments is to study the correlation between the analogical retrieval strategies and the creative properties (as defined in section 2.2) of the generated diagrams. The creative properties in which we are interested are usefulness and novelty. A set of 25 UML class diagrams were used as problems during these experiments, each problem comprises, on average, four objects.

During these experiments we used six candidate selection strategies (see table 5.18) for reuse through analogy. When semantic retrieval is used as part of one of these strategies, only ten (the first ten cases found by our retrieval algorithm) cases are used for ranking, otherwise it is used the whole case base. After the ranking phase, the most promising candidate is submitted to the analogy module. Afterwards, the output of the analogy module (an UML class diagram) is evaluated both on usefulness

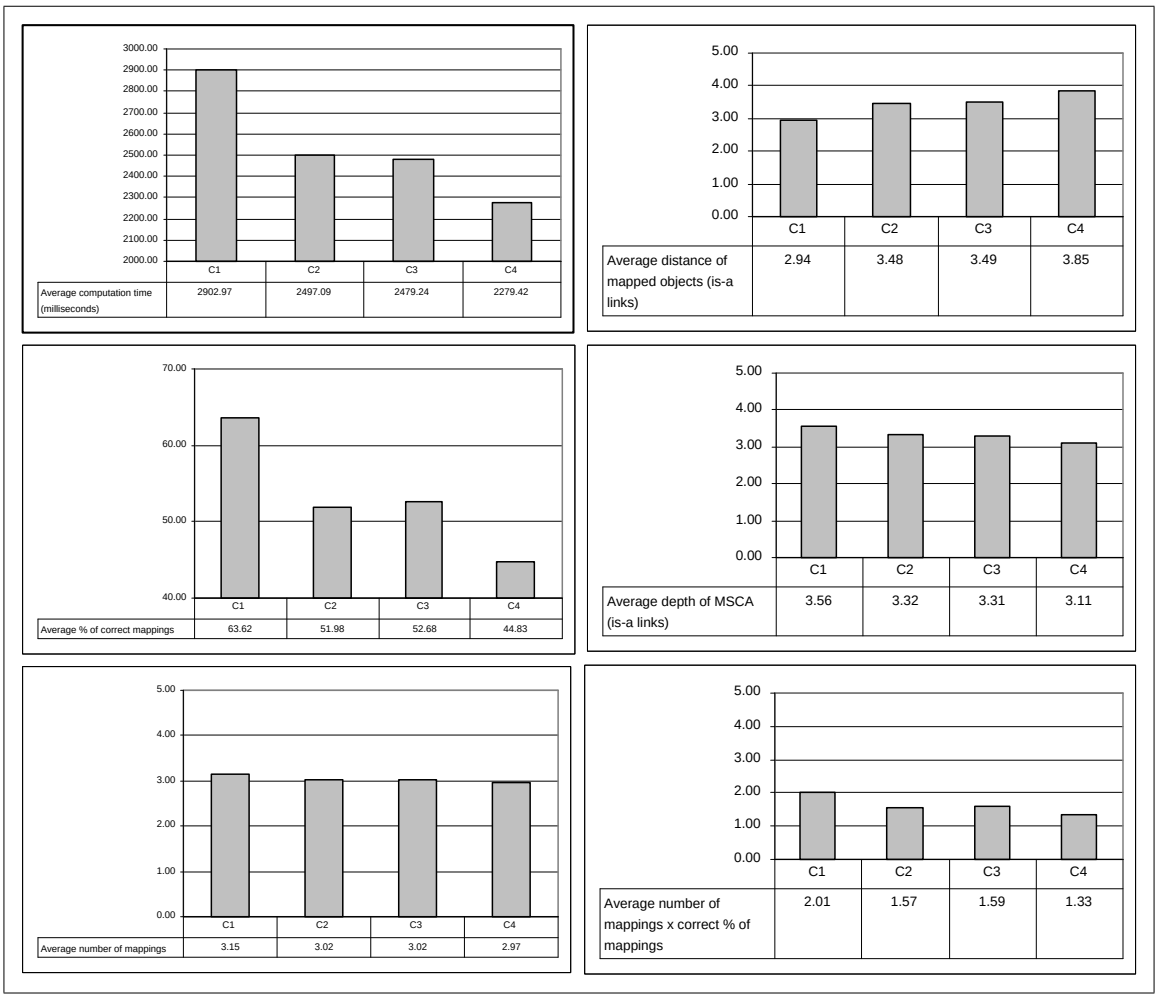


Figure 5.11: Experimental results obtained for the first five solutions presented by REBUILDER, and for the three problem sets (average of values for P20, P50 and P80).

and novelty. For these experiments we use the object-based mapping algorithm, since this algorithm has provided better results in previous work (see previous subsections).

Novelty is assessed by the similarity metric used in case retrieval. We assess the novelty of a solution (the output of the analogy module) using the similarity with the cases in the case base. Three different similarity values are computed using the generated solution:

N1: the average similarity between the generated solution and the cases in the case library.

N2: the similarity value between the solution and the most similar case in the case library.

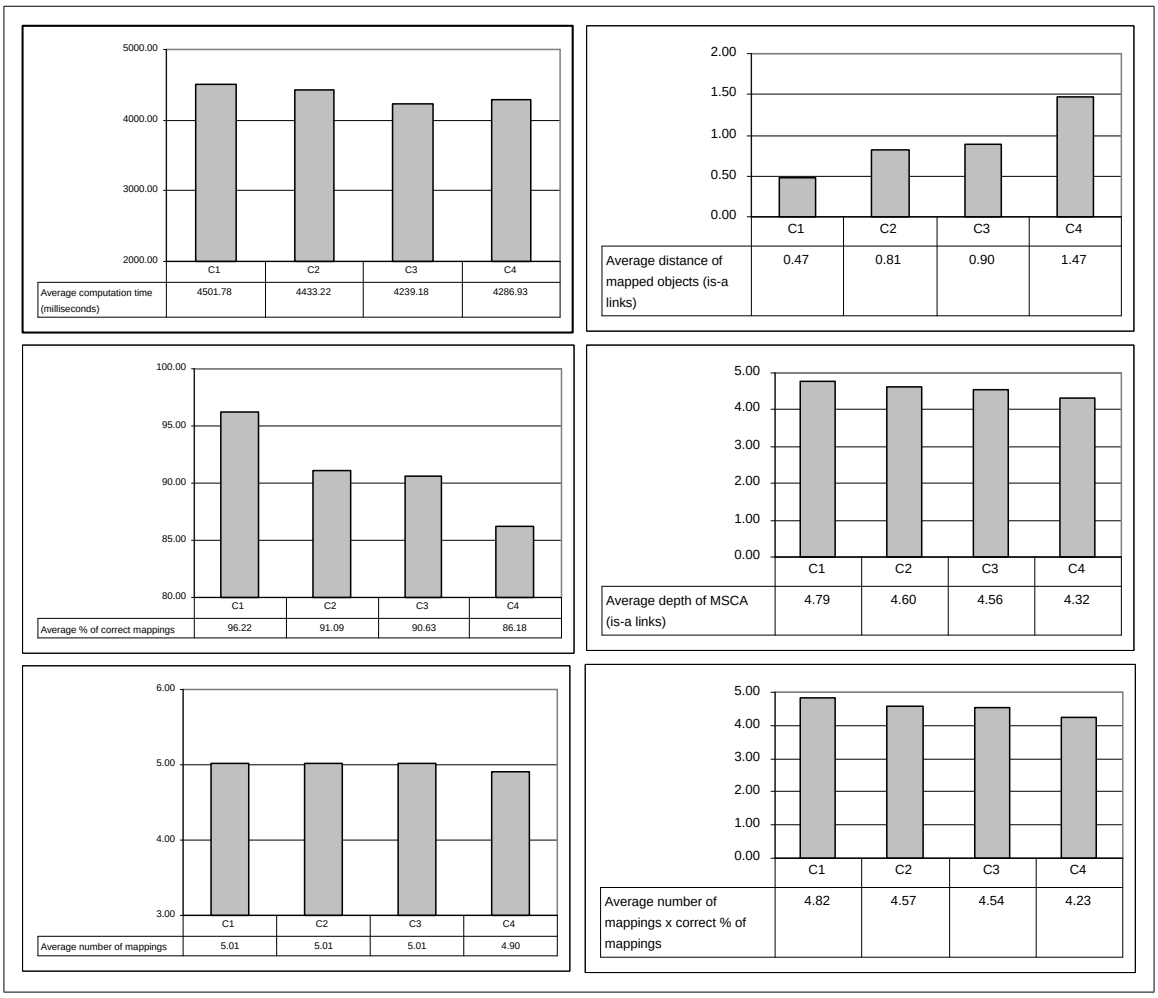


Figure 5.12: Experimental results obtained for the first solution presented by REBUILDER, and for the three problem sets (average of values for P20, P50 and P80).

N3: the similarity value between the solution and the source case chosen for establishing the analogy with the target case.

Since the similarity metric, returns values between 0 and 1, where 1 is an identical match, we are interested in situations where this value is low. The results of this experiment are presented in figure 5.13.

Usefulness is evaluated through human judgement of the generated diagrams. Two human judges evaluated these diagrams, identifying:

U1: the percentage of target objects that were mapped to a source case in which the knowledge transferred (methods and attributes from the source case to the target case) are incorrect or useless.

Table 5.18: The retrieval strategies implemented in REBUILDER.

Strategy	Retrieval	Ranking
1	Semantic	Semantic and Structural
2	Semantic	Structural (independence measure)
3	Semantic	Structural
4	No	Semantic and Structural
5	No	Structural (independence measure)
6	No	Structural

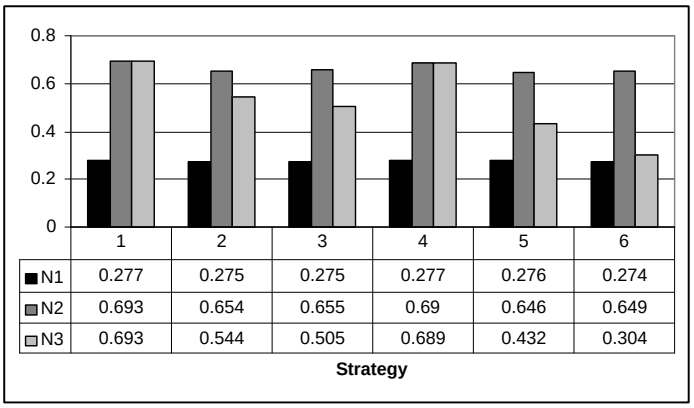


Figure 5.13: Novelty experimental results for analogy generated solutions.

U2: the percentage of target objects that were mapped to a source case in which the knowledge transferred (methods and attributes from the source case to the target case) are partially incorrect.

U3: the number of objects that were transferred to the target diagram and that were considered incorrect or useless.

These values are presented in figure 5.14. We are aware of the subjectivity of the results for these experiments, especially in which concerns to the judgement of usefulness. We feel that a more useful evaluation is through a case study in a real software production environment where the tool is judged by its users. Despite this fact we still think worth to present our evaluation.

As can be seen in figure 5.13 the average distance of the solution in relation to the rest of the case library is practically the same. This result may be due to the fact that the cases library is sparse, in the sense that different domains are modelled by the cases present in the case library. So whatever the solution, on average, it will

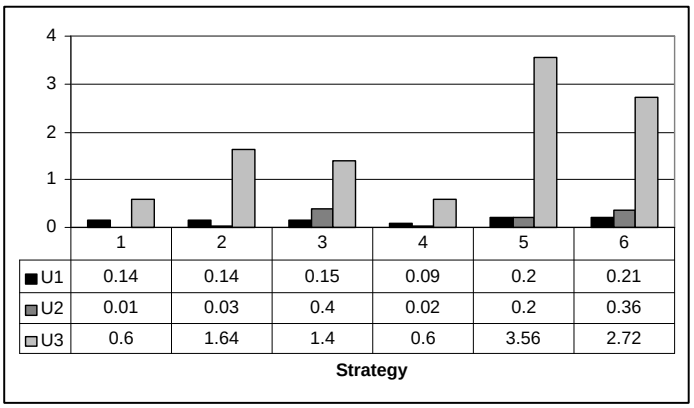


Figure 5.14: Usefulness experimental results for analogy generated solutions.

tend to be dissimilar to the case library.

Observing the value of similarity between the solution and the most similar case (N2), we see that strategy one and four present the highest values, which makes sense since this metric contains a semantic component. In respect to N3, we can see that strategy five and six obtain solutions that are dissimilar to the source case used for establishing analogies. These values were expected, taken into consideration that neither of these strategies uses semantics during the retrieval or ranking phases, and that the similarity metric does. Thus, these strategies are able to generate more novel solutions than the other strategies, which are using semantics to retrieve and rank source cases.

Figure 5.14 presents the values obtained during the evaluation of the generated solution usefulness. We remind the reader that these values represent situations that were considered wrong or useless, this means that lower values correspond to solutions that are more useful since they require less corrections from the software designer. As we can observe, strategies five and six seem to generate the diagrams which require larger corrections. These high values may be due to the fact that the knowledge transfer scheme that was implemented simply transfers objects from the source domain into the target domain without transforming the source object into an analog in the new target domain. If these source objects are transferred into completely different domains they will usually be considered incorrect.

These results suggest that semantics should be considered during the retrieval phase in order to obtain the most useful diagrams, that is, diagrams that require

Table 5.19: The configurations used in the design composition experiments. BSCC - Best Set of Cases Composition, BCC - Best Case Composition.

Strategy	Generation Method	Threshold
C1	Analogy	0.25
C2	Analogy	0.5
C3	Analogy	0.8
C4	Design Composition (BSCC)	5
C5	Design Composition (BSCC)	10
C6	Design Composition (BSCC)	15
C7	Design Composition (BCC)	5
C8	Design Composition (BCC)	10
C9	Design Composition (BCC)	15

fewer corrections. On the other hand, novel analogies tend to occur when the ranking phase is based on structural properties. Using a retrieval strategy that combines both aspects can be a good solution for analogy retrieval. Examples of this are strategies two and five, which are able to maintain a compromise between strategies one and four, which are more accurate but less aimed to generate novel solutions, and strategies three and six that are capable of generating more novel solutions but with an increase in the number of errors in the class diagrams.

5.4 Design Composition Experiments

This section describes the experiments performed with the design composition module. These experiments have several goals: compare design composition with analogy, compare the design composition strategies, and identify the best threshold values for the design composition strategies.

The design composition mechanism uses a parameter that establishes if an object can be mapped or not with a problem's object. If a mapping candidate object is conceptually closer to the problem's object than the threshold value, it implies that this object can be mapped with a problem object. To assess the conceptual distance REBUILDER uses WordNet *is-a* links, so the threshold value is a maximum mapping distance.

These experiments use the 25 problem set, with nine solutions generated in each

Table 5.20: Experimental results obtained from test users. BSCC - Best Set of Cases Composition, BCC - Best Case Composition.

	Computation Time (seconds)	Transferred Objects	Mapped Objects	Used Cases	Wrong Objects	% of Wrong Objects	% of Mapped Objects
C1	28.48	8.00	1.56	1.00	2.16	16.54%	35.73%
C2	28.44	8.08	1.48	1.00	2.28	17.80%	34.13%
C3	26.64	7.36	1.16	1.00	1.52	11.98%	26.93%
C4	52.48	5.00	2.88	1.32	1.64	11.86%	64.60%
C5	59.68	5.84	3.12	1.28	2.72	21.57%	71.07%
C6	60.48	5.84	3.12	1.28	2.92	22.93%	71.07%
C7	72.08	18.20	3.08	4.40	6.96	29.18%	70.27%
C8	73.20	18.52	3.08	4.36	7.16	29.24%	70.27%
C9	74.96	18.52	3.08	4.36	7.20	29.44%	70.27%

problem run, one for each configuration (table 5.19 defines the configurations). The problems and their respective solutions were then presented to test users (software designers and software engineers) for evaluation. Six test users were inquired for evaluation of the solutions, giving their evaluation on the number of objects that they considered inadequate or incorrectly defined, regarding the problem that it was supposed to solve. Most of the designers made this judgment based on what they would delete from the suggested solution in order to solve the problem under attention. The results obtained are presented in table 5.20. The values presented are the average values for the 25 problems.

Configurations using analogy (C1 through C3) exhibit the best time performance. While the Best Case Composition (BCC) configurations (C7 to C9) show the worst performance. In between, we have the Best Set of Cases Composition (BSCC) configurations (C4 to C6). All BCC and BSCC configurations increase the computation time with the increase in the threshold, which is expected, since the range of possible mappings increases, and with it the computation time needed to generate them and to evaluate them.

Concerning the number of transferred objects the BCC configurations have the highest values. BCC uses many cases to generate solutions, which implies the transference of many non-mapped objects, not all of them useful as the number of wrong objects illustrates. The analogy configurations have an unexpected high number of transferred objects, which can be explained by the low number of mappings that

are performed. The objects that are not mapped, are then transferred. The BSCC achieves a low number of transferred objects, but it is compensated by a high percentage of mapped objects. There is an interesting finding in the design composition configurations (configurations C5 and C6) with thresholds of 10 and 15 they generate the same number of transferred cases, the same happening with configurations C8 and C9. This might indicate that after a certain threshold value the established mappings are the same, reaching a saturation value.

Configurations C5 and C6 achieve the best results for the number of mapped objects, closely followed by C7, C8 and C9. Though BSCC configurations use less cases than BCC configurations, they are able to have a higher efficiency (or accuracy) in which concerns mapping problem objects. Only by restricting this mapping process (through the lowering of the threshold value) the number of mapped objects decreases (see configuration C4). The analogy configurations have the worst performance. Note that the values for the percentage of mapped objects corroborate these observations. In the analogy configurations the number of mapped objects is proportional to the decrease of the threshold (remember that the analogy threshold has a different meaning from the one for the design composition threshold - higher values mean more constrained mappings).

Naturally, analogy configurations use only one case, since they are able to work with only one instance. In regard to the design composition configurations, the BSCC uses less cases than BCC, achieving better mapping results and with fewer wrong objects. This clearly indicates that the BSCC mechanism is more efficient and accurate than the BCC strategy. Configurations with threshold values more restrictive use more cases, which is coherent, since they are not able to map all the problem objects, and have to look for other pieces of cases to get these mappings.

The configuration that has the lower percentage of wrong objects is C4. This configuration uses the BSCC strategy to generate new solutions efficiently and is able to achieve the solutions with fewer mistakes. Analogy configurations can also achieve good results. The BCC generates the worst results, which can be partially explained by the high number of transferred objects.

In conclusion, BSCC configurations achieve the best tradeoff between computation time, number of wrong objects and number of mapped objects. A good threshold

value for design composition strategies is 10, which works for both strategies. Further experiments on the threshold value should be performed to pinpoint the best threshold value.

5.5 Design Patterns Experiments

This section describes the experiments performed to evaluate the performance of the design patterns module. We used a case base of 60 DPA cases, each case representing an application of a software design pattern to an UML class diagram (we used the design case base of 60 designs to generate the DPA cases). Each DPA case was generated from a different class diagram. For these experiments, five software design patterns were used, the names of the patterns accordingly to [Gamma et al., 1995] are:

- Abstract Factory.
- Builder.
- Composite.
- Singleton.
- Prototype.

Each of these patterns are implemented in REBUILDER along with the participants definition and operators (see [Gamma et al., 1995] for details on these patterns).

We also defined 25 test class diagrams to evaluate the precision of the retrieval mechanism. These diagrams comprise three to five objects and do not have methods or attributes. For each test diagram the algorithm retrieved 15 DPA cases, which were then evaluated. The evaluation consisted in defining the patterns and participants that were chosen correctly. The results are presented in figure 5.15.

The precision results show that the retrieval mechanism for this set of problems achieves 76% of correct selected DPA cases with a retrieval set size of 3 (cumulative result), which is in our opinion a good indicator. As expected, the non-cumulative results degrade with the increase in the ranking of retrieved cases. This also indicates

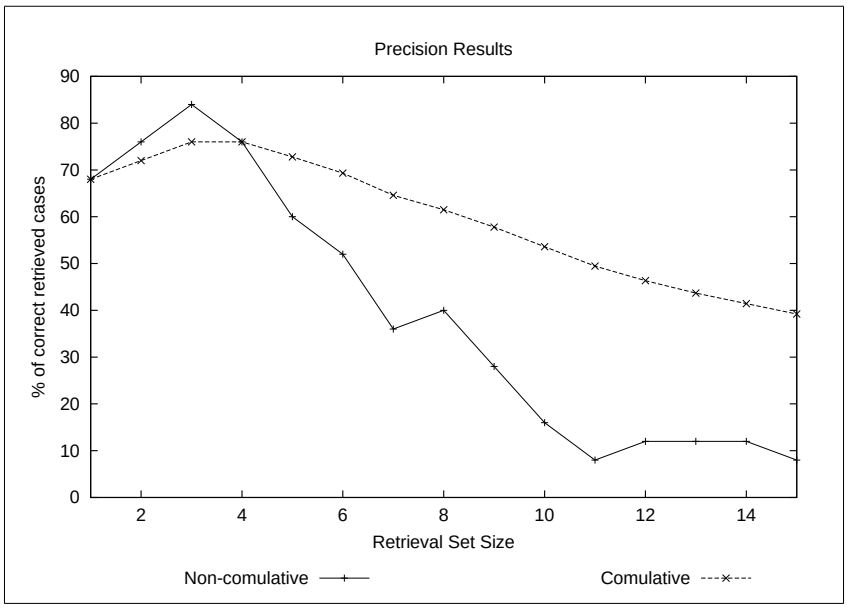


Figure 5.15: The precision values for the DPA retrieval algorithm.

that the similarity metric that was used to rank the retrieved cases is performing as desired, choosing the best cases for the higher places in the ranking. The cumulative values show that the precision ranges from 76% (retrieval set size of 3 and 4), to 39% (retrieval set of size 15).

5.6 Verification Experiments

This section describes the experimental work performed to evaluate the verification mechanism. We tested the verification for analogy and design composition. Within the design composition we used both strategies: best case composition and best complementary cases composition (also called best set composition).

For each combination of the method for solution generation (Analogy, Design Composition - Best Case, and Design Composition - Best Set) by type of object considered (Relation, Attribute or Method) we generated an experiment with the same 25 test problems and then gathered the data, which was then analyzed by a software engineer.

Using verification, the number of wrong objects decreased by 22% in analogy and by 56% in design composition. The detailed results are presented in table 5.21, the

Table 5.21: Experimental results obtained for the verification mechanism. A - Attributes; M - Methods; R - Relations.

	Analogy			Best Case Composition			Best Set Composition		
	A	M	R	A	M	R	A	M	R
EO	0	0	2	76	22	91	70	26	42
WO	0	2	7	77	116	127	70	135	58
IM (%)	-	0%	29%	99%	19%	72%	100%	19%	72%

key for the table is:

EO: objects eliminated that were considered wrong by human evaluators, in which the system learned new failure cases.

WO: total number of wrong objects.

IM: improvement with verification (in percentage).

5.6.1 Effect of Verification in Analogy

Figure 5.16 presents the cumulative number of wrong objects, with and without verification. The X-axis represents the 25 problems that were used. In this case, the presentation order of these problems is relevant, since the system learns new verifications from each problem it solves. We are only considering: relations, attributes and methods. In this scenario, where solutions are generated by analogy, there are no attribute objects considered wrong by the designers. Notice that the solutions generated by analogy result into few wrong objects compared to the ones created by design composition. This happens because design composition combines parts of different cases, thus generating some inconsistencies. The verification mechanism did not improve the number of wrong methods, and decreased in 29% the number of wrong relations.

5.6.2 Effect of Verification in Design Composition

Figures 5.17 and 5.18 show the experimental results for the effect of verification in design composition. These results show a major improvement in the solution quality,

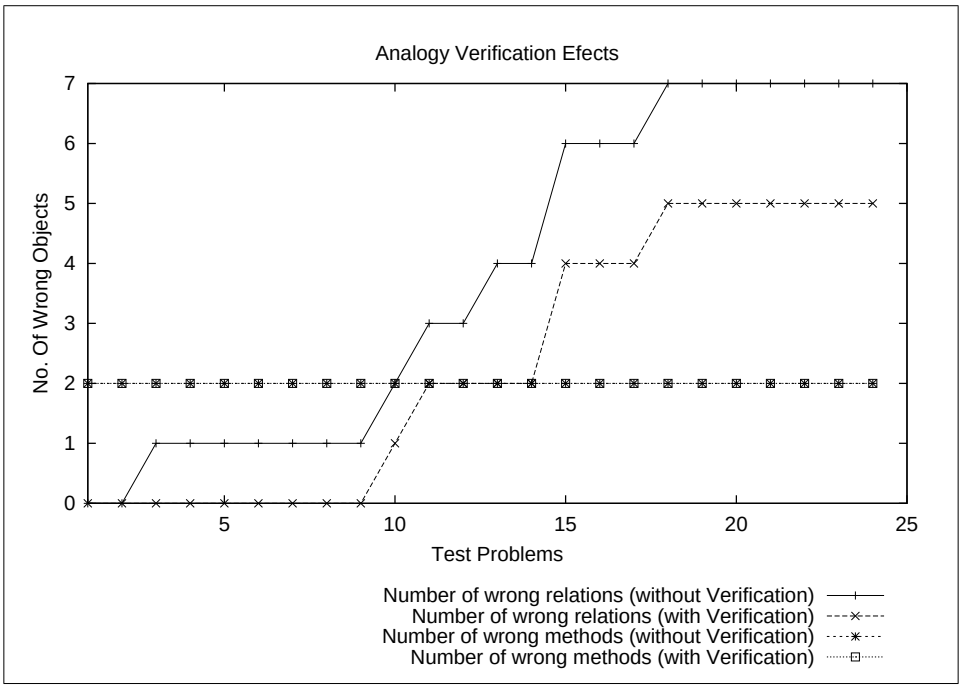


Figure 5.16: Effects of the verification process in analogy generated solutions.

in particular in the attributes and relations. The system can effectively drop the number of wrong objects in generated solutions, presenting to the designer better solutions.

Table 5.19 presents the total number of verification cases generated and stored by the verification module, and also the number of questions asked to the designer. Notice the difference between the number of verification cases and questions asked to the designer. Most of the verification cases come from the design case library (from the design cases) and from WordNet. Since the system learns these new verification cases, the tendency in the evolution of the number of questions asked to the designer is to decrease, especially if the designer is working in the same domain.

5.7 Learning Experiments

The experimental work performed using the learning module (see section 4.6) has the goal of determining the characteristics of the implemented criteria, mainly in two aspects: computational time and evolution of the case base competence. The next subsections explore both issues.

We used the previous KB and the 25 problem set. It should be noted that despite

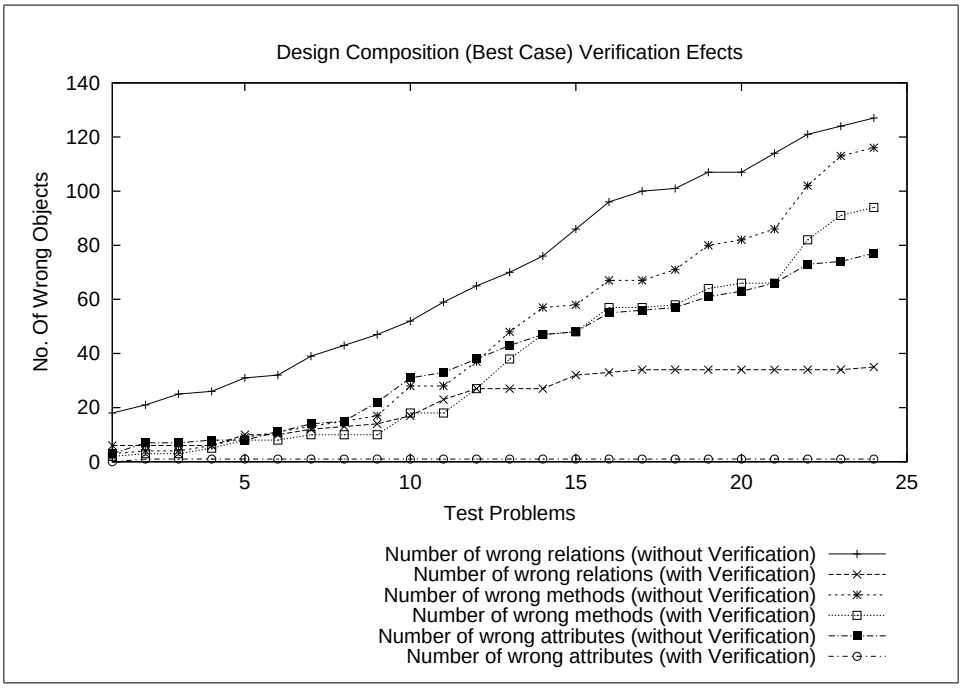


Figure 5.17: Effects of the verification process in the design composition (Best Case) generated solutions.

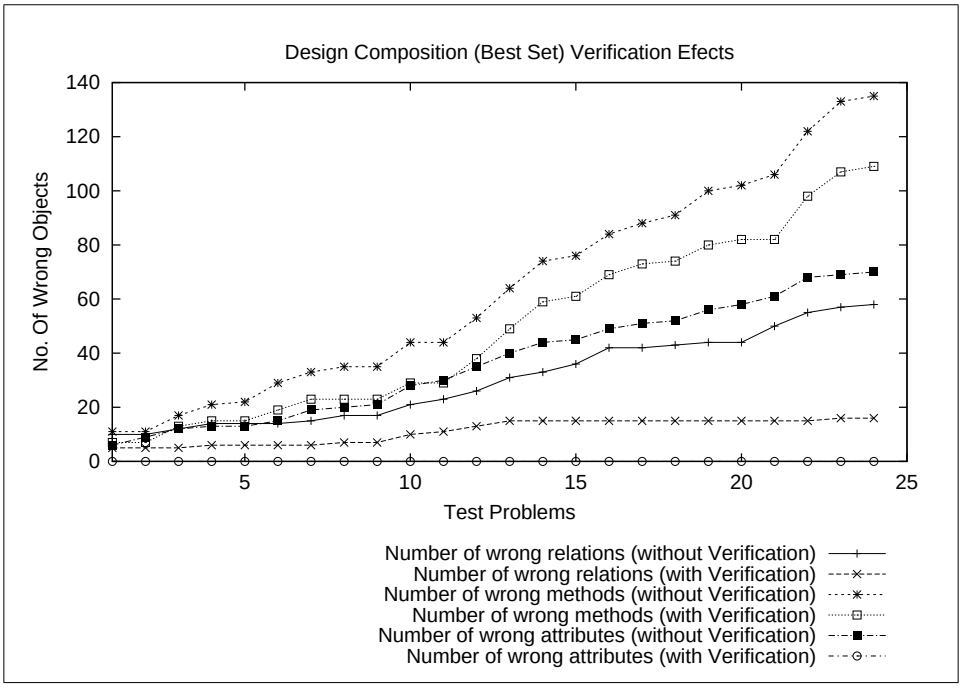
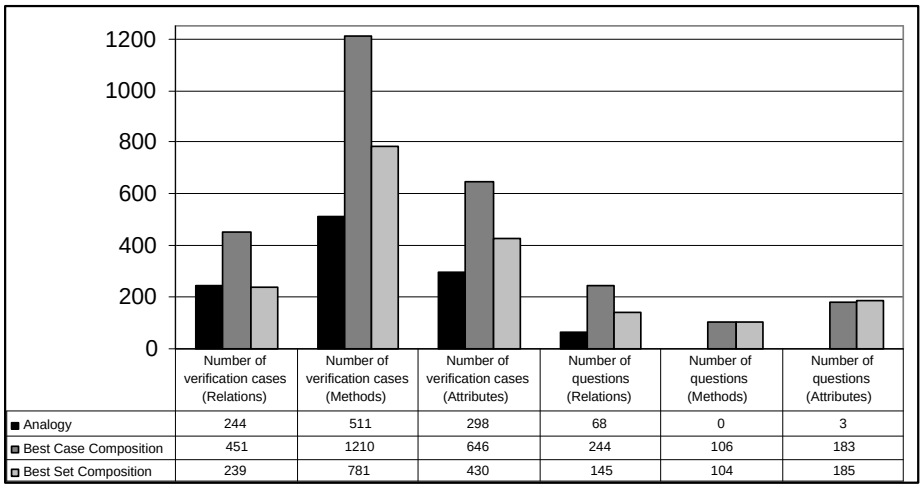


Figure 5.18: Effects of the verification process in the design composition (Best Set) generated solutions.



	Number of verification cases (Relations)	Number of verification cases (Methods)	Number of verification cases (Attributes)	Number of questions (Relations)	Number of questions (Methods)	Number of questions (Attributes)
Analogy	244	511	298	68	0	3
Best Case Composition	451	1210	646	244	106	183
Best Set Composition	239	781	430	145	104	185

Figure 5.19: Number of verification cases and questions asked to the designer.

the problem set has defined case domains, when solving a problem, the retrieval system can use problems from any domain. The retrieval algorithm can do this due to the indexing structure (WordNet), which connects all synsets.

5.7.1 Computational Time Efficiency

Although being in a certain sense irrelevant to test the time performance of all these algorithms from a research point of view, we think that for application purposes this is of great relevance. It would be interesting to take further these experiments, especially when dealing with case bases that, when in use, store a huge number of cases, or when we want to study scaling properties of these strategies. Along the description of these experiments we use abbreviations to refer to the strategies:

FDC: Frequency Deletion Criteria.

SC: Subsumption Criteria.

FtDC: Footprint Deletion Criteria.

FtUDC: Footprint Utility Deletion Criteria.

CC: Coverage Criteria.

CAC: Case-Addition Criteria.

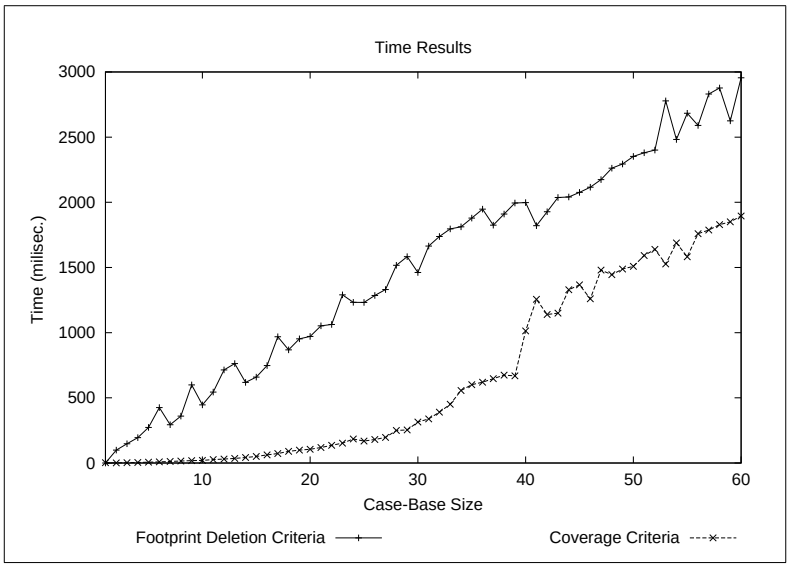


Figure 5.20: Computation time results for criteria FtDC and CC.

RCCNN: Relative Coverage and Condensed NN.

RPM: Relative Performance Metric.

CGC: Competence-Guided Criteria.

The results are presented in figures 5.20 and 5.21, and show that there are three different groups of criteria, regarding computational time and scalability. The first group comprises FtDC and CC (see figure 5.20), which have a very different performance comparatively to the other criteria. Their computation time grows in a linear way, but with a much bigger multiplication factor. In particular, FtDC performs worst than CC, at least for the 60 case library that was used. The second group contains CAC, CGC, FDC, and FtUDC, which in the experiments grow linearly, but with a much lower multiplication factor than the first group. FDC and FtUDC are very regular, contrasting with CAC and CGC, which had a slightly exponential growth between iteration 25 and 33. We think that this is due to the sequence of cases that were used. It was possibly a set of cases that needed more computational time to run the algorithm. Criteria SC and RPM are the third group, which have very low computational times (bellow one millisecond most of the times). SC is very fast, because among the 60 cases there are no case subsumptions. These results are due to the subsumption criteria defined, and to the huge search space, from which we have selected 60 'points' close enough to have similarities, but not enough to be subsumed.

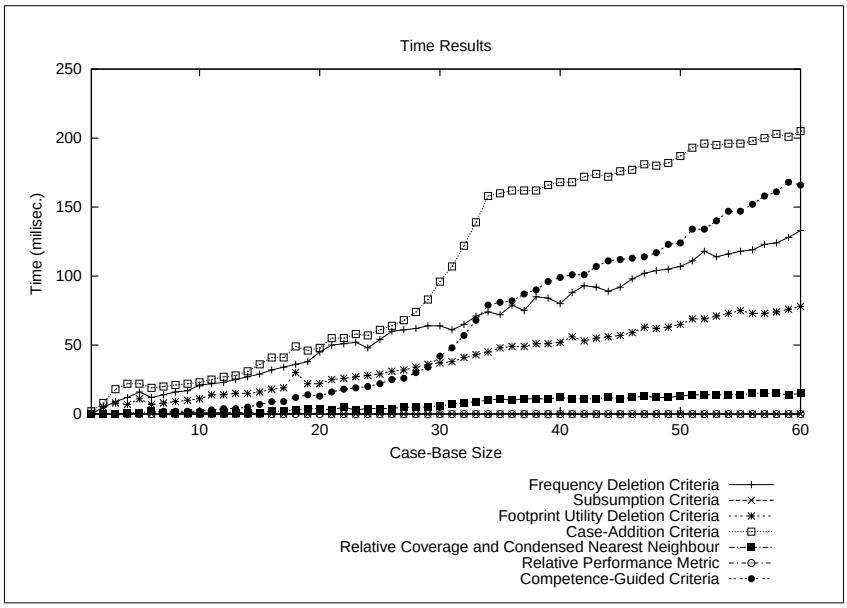


Figure 5.21: Computation time results for the other criteria (FDC, SC, FtDUC, CAC, RCCNN, RPM, and CGC).

These are indicative results and not definitive ones, because these results are always limited by the cases that were used for testing. Within our case base, cases populate a small portion of the design space, even if we take into account the adaptation mechanisms. These tests were performed on an Intel Pentium III at 600 MHz with 256 Mbytes of memory running Windows 2000.

5.7.2 Case Base Competence

The second experiment comprises the variance of the case base size, using the CBM strategies to define which cases maximize the case base competence. We have used case bases of six sizes: 10, 20, 30, 40, 50 and 60. The last one comprises all the possible cases, so it is a degenerate situation and there is no need to apply the strategies. The case base of size 60 is used as the benchmark for the case base. For each criteria we evaluated the case base competence using the test problems. For each test problem we ran the retrieval and the adaptation (design composition) algorithms and evaluated the suggested designs, as being relevant or not. For retrieval, we considered a retrieved case relevant if it belongs to the first five that are more similar to the problem. We defined similar as having the same package synset, and the same objects (same synset) except one. In the adaptation, only one generated solution was inspected, using the same solution evaluation criteria as in retrieval.

Table 5.22: Competence results computed using the CBM strategies, without adaptation.

CB Size	FDC	SC	FtDC	FtUDC	CC	CAC	RCCNN	RPM	CGC
10	24%	24%	4%	24%	24%	4%	24%	24%	24%
20	36%	36%	4%	36%	36%	8%	36%	36%	36%
30	48%	48%	8%	48%	48%	12%	48%	48%	48%
40	48%	48%	16%	48%	48%	36%	48%	48%	48%
50	48%	48%	40%	48%	48%	36%	48%	48%	48%
60	48%	48%	48%	48%	48%	48%	48%	48%	48%

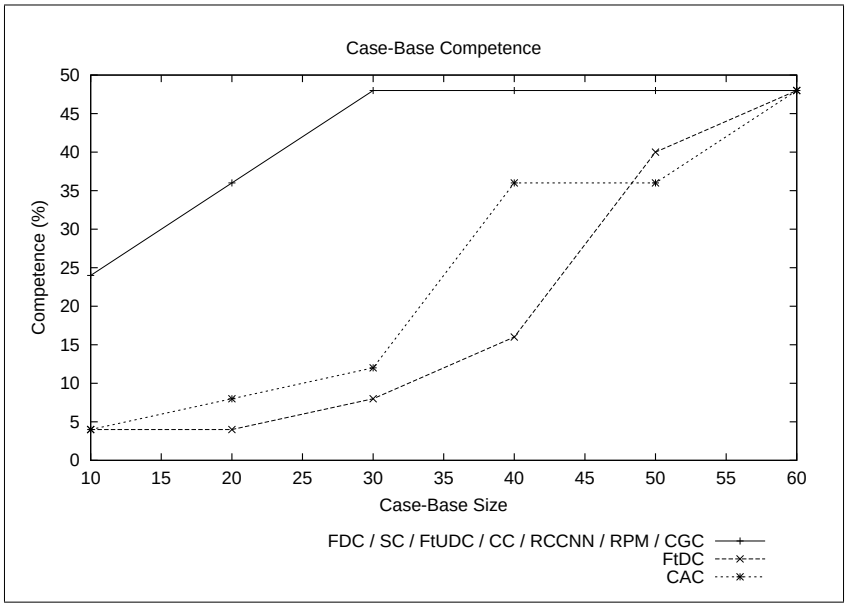


Figure 5.22: Competence results computed using the CBM strategies, without adaptation.

From the results presented in tables 5.22 and 5.23 (see figures 5.22 and 5.23 for a graphical representation), it can be concluded that the adaptation mechanism makes a big difference in the evaluation of the case base competence. Even with ten cases, the case base with adaptation has a higher competence than just with retrieval. An explanation for this is that the solution space (UML class diagram space) is huge and the 60 cases are only a very small portion of it. The normal situation is the target problem to be different from the cases in the case base, which enables retrieval to find a similar case, but not similar enough to solve it. The adaptation mechanism is robust enough to use a retrieved case to solve the target problem.

Analyzing the results in which concerns to the different case base maintenance policies and using only these experiments, it is obvious that most of them lost performance when reducing the number of cases in the case base. Despite that most criteria only originate case bases with lower competence when they lose more than

Table 5.23: Competence results computed using the CBM strategies, with adaptation.

CB Size	FDC	SC	FtDC	FtUDC	CC	CAC	RCCNN	RPM	CGC
10	92%	92%	68%	68%	92%	92%	92%	92%	92%
20	92%	92%	72%	68%	92%	92%	92%	92%	92%
30	96%	96%	92%	92%	96%	92%	96%	96%	96%
40	100%	100%	96%	100%	100%	96%	100%	100%	100%
50	100%	100%	96%	100%	100%	96%	100%	100%	100%
60	100%	100%	100%	100%	100%	100%	100%	100%	100%

50% of the cases. This may indicate that there are redundant cases in the case base, and it seems that most of the strategies can identify and remove them from the case base.

There is another important aspect, which is the application of these criteria to this case representation. We have the opinion that the subsumption definition influences the applicability of some strategies. For instance, we have defined in the implementation of the criteria, that when there are draws between cases, the criteria should choose cases driven by the frequency of case usage (except when the CBM strategy explicitly states what to do). Since most of the cases are not subsumed, there are a lot of draws, which makes this default selection form to be used most of the times. This is why most of the criteria have a similar performance comparing to the FDC, which is a pure frequency-based approach.

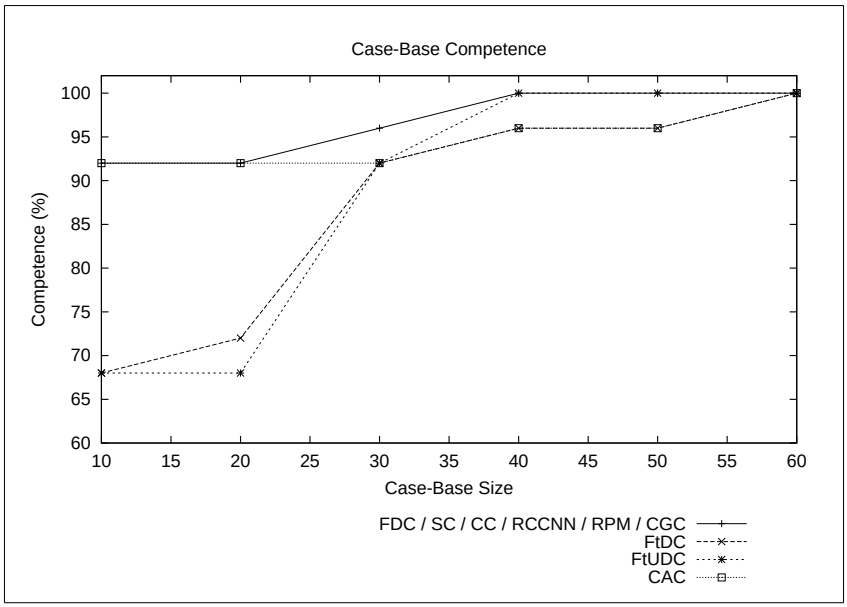


Figure 5.23: Competence results computed using the CBM strategies, with adaptation.

Table 5.24: The definitions for the context configurations.

Configuration	Object Attributes	Neighbor Objects	All Objects
C1	Yes	No	No
C2	No	Yes	No
C3	Yes	Yes	No
C4	No	No	Yes
C5	Yes	No	Yes

5.8 Word Sense Disambiguation Experiments

This section describes two experiments performed with the Word Sense Disambiguation (WSD) methods incorporated in REBUILDER. First we describe preliminary experiments, which were followed by a more elaborated experiment.

5.8.1 First Experiments with WSD

This subsection presents two studies of one of the proposed WSD methods - conceptual distance version 1 (see subsection 4.7.7). One study comprises choosing the best context configuration for object disambiguation, and the second one showing the best semantic distance. The KB we used for tests comprises the previous case library with 60 cases. The goal is to disambiguate each case object in the KB. After running the WSD method for each object we have collected the selected synsets, which are then compared with the synsets assigned by a human designer. The percentage of matching synsets determines the method accuracy. To study the influence of the context definition in the disambiguation accuracy, we considered five different combinations (see table 5.24).

The accuracy results we obtained are presented in Table 5.25. These values show that the best result is reached by configuration C4, and that configuration C5 presents slightly worst results than C4. This is due to the abstract aspect of some attributes, which introduce difficulties on the disambiguation method. For example, if one of the attributes is *name*, it will not help in the disambiguation task, since this attribute is used in many objects and is a very ambiguous one. Notice that configuration C1 is only able to select a synset for 89.33% of objects.

In these experiments we used three different semantic distances. These distances

Table 5.25: The accuracy values for the context configurations.

	C1	C2	C3	C4	C5
Accuracy	60.19%	68.47	68.79	71.18%	71.02%
% of Synsets Found	89.33%	100.00%	100.00%	100.00%	100.00%

Table 5.26: The accuracy values for the different semantic distances.

Accuracy	C4	C5
S_1	69.27%	69.11%
S_2	71.18%	71.02%
S_3	64.97%	65.13%

are: the *is-a* links of WordNet alone (S_1), the *is-a*, *part-of*, *member-of* and *substance-of links* (S_2), and S_3 which is similar to the semantic distance as defined in subsection 4.2.2. The previous results (shown in table 5.25) are obtained with S_2. A combination of these three distances and the best context configurations (C4 and C5) were used to study the influence of the semantic distance to the accuracy of the WSD method. Results are shown in table 5.26.

Experimental results show that semantic distance S_2 produces the best accuracy values, followed by S_1, and finally S_3. Context configuration C4 continues to get better results than C5, except for semantic distance S_3.

5.8.2 Comparison of WSD Methods in REBUILDER

After generating the experiments presented in the previous subsection with the WSD method developed, we wanted to test other WSD methods. Our main goal was to find the best WSD method for REBUILDER, which is defined along two dimensions: computational efficiency (computation time) and disambiguation accuracy. Computational efficiency is important because each time the designer uses a cognitive action, the software objects must have synsets associated to them. In case of a class diagram with many objects, the system must be fast enough to be usable, otherwise the designer will not call the system's reasoning capabilities. Disambiguation accuracy, because the retrieval and all the other cognitive processes depend on the assignment of the right synsets, otherwise the system will return useless diagrams to the designer. We have tested methods (see section 4.7) along: computation time and

result precision.

Another important aspect of REBUILDER, is that, it is not a common natural language application, in the sense that it does not work with natural language paragraphs or sentences, but with loose words associated to software objects. This made us choose two different experiments: one specific to REBUILDER, and another similar to previous WSD studies. In the first experiment we have used a case base with 60 cases, and disambiguated all cases and objects. The other experiment was performed using SemCor 1.7, built from the Brown corpus and distributed with WordNet 1.7.1[3]. The next two subsections explain in detail these two experiments showing the quality achieved with each method, while the third subsection reports the time results achieved by each method.

Experiment 1 - Case Base

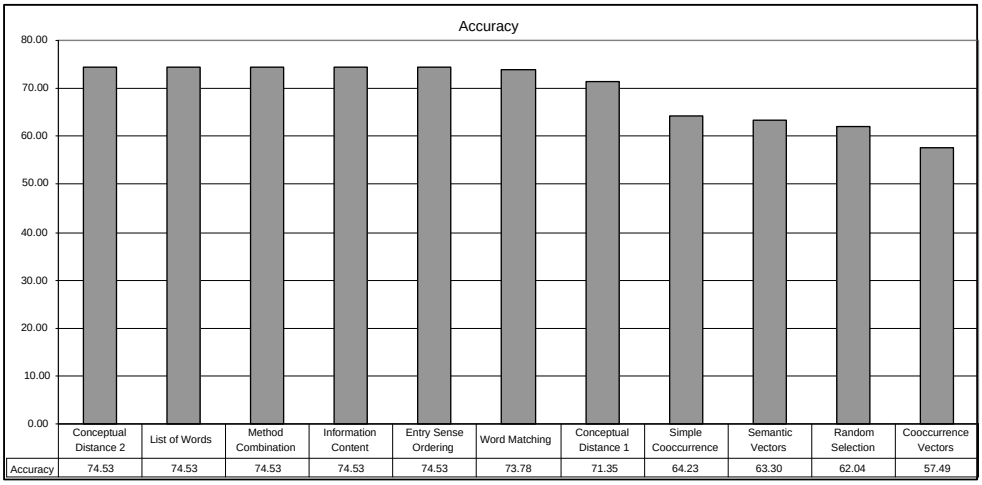
The first experiment was performed using the case library with 60 cases, each case describing a software design. The goal was to disambiguate all objects in the case library. Each object of each case were disambiguated using only the names of the objects in the same diagram as the context words. Then the target word (the object's name) was disambiguated using all WSD methods. This experiment has two variants: one where all the case objects need to be disambiguated, simulating the situation where the entire diagram needs to be disambiguated (Diagram Disambiguation); and another where each case object is disambiguated with all the other case objects already disambiguated [4] (Object Disambiguation). The disambiguation results that were obtained are shown in figures 5.24 and 5.25.

In the Diagram Disambiguation five methods have achieved identical results, showing a good accuracy. In this experiment all diagram objects were disambiguated, including the non ambiguous[5] objects, which are about 46%. In the Object Disambiguation the results are only about ambiguous words (average sense number by word is 4.30). Figure 5.25 shows that the Entry Sense Number method is the best one, followed by Word matching, and then by Information Content and List of Words. As

[3]Available at http://www.cogsci.princeton.edu/∼wn/

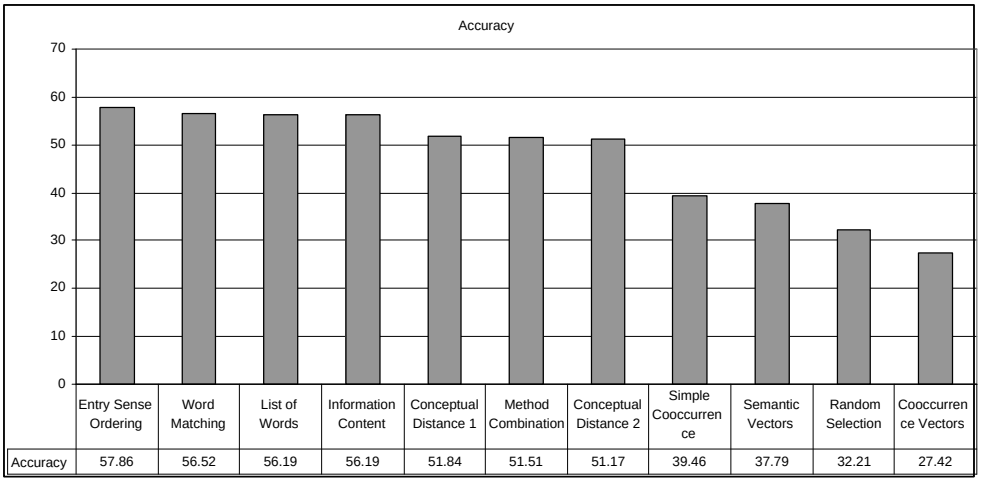
[4]We have manually disambiguated each object.

[5]Objects which the name has only one candidate synset, so they are trivial to disambiguate, that is why they are called non ambiguous.



	Conceptual Distance 2	List of Words	Method Combination	Information Content	Entry Sense Ordering	Word Matching	Conceptual Distance 1	Simple Cooccurrence	Semantic Vectors	Random Selection	Cooccurrence Vectors
Accuracy	74.53	74.53	74.53	74.53	74.53	73.78	71.35	64.23	63.30	62.04	57.49

Figure 5.24: Experimental results obtained for the diagram disambiguation.



	Entry Sense Ordering	Word Matching	List of Words	Information Content	Conceptual Distance 1	Method Combination	Conceptual Distance 2	Simple Cooccurrence	Semantic Vectors	Random Selection	Cooccurrence Vectors
Accuracy	57.86	56.52	56.19	56.19	51.84	51.51	51.17	39.46	37.79	32.21	27.42

Figure 5.25: Experimental results obtained for the object disambiguation.

a comparison baseline we used the random selection method, which selects a synset randomly from the possible ones. All the methods exceed this baseline except the Cooccurrence Vectors method. The gains on the other methods, in relation to the baseline, go up to 20% in the Diagram Disambiguation, and 80% in Object Disambiguation.

Experiment 2 - SemCor

After evaluating the WSD methods in a specific environment (or set of domains) we wanted to see which were the results of these methods in more general situations, and with more ambiguous words. For this, we used SemCor 1.7 of the Brown Corpus, which is part of the Brown corpus tagged with WordNet senses. 47 words have been selected, mainly from the literature on WSD, to be used as target words (the

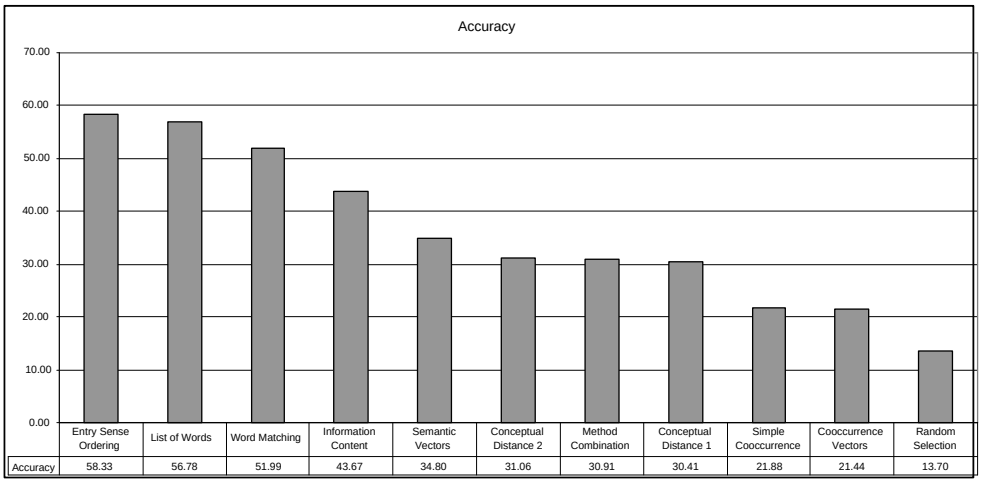


Figure 5.26: Experimental results obtained for the SemCor 1.7 disambiguation.

average sense number by word is 7.30): *star, cone, bass, bow, taste, interest, issue, duty, sentence, slug, bank, doctor, nurse, age, art, car, child, church, cost, work, head, line, device, direction, reader, core, right, position, deposit, hour, path, view, antenna, trough, tyranny, figure, institution, crown, drum, pipe, coverage, execution, min, interior, campaign, output, drive*. We used 186 files from SemCor comprising about 15000 sentences. Whenever one of the target words appears in a sentence it is disambiguated using the words in the same sentence as context. The context words had the correct synsets attributed (the ones provided by SemCor). All the WSD methods were used, and for each disambiguation a comparison with the SemCor synset was performed yielding the method accuracy.

The results achieved in this experiment are presented in figure 5.26. Comparing these results with the Object Disambiguation ones, it can be seen that some methods maintain similar performances (List of Words and Entry Sense Ordering), while all the other methods lose accuracy. It is important to note that the baseline (Random Selection) decreased from 32% to 14%, due to the high number of senses by word.

Experiment 3 - Computation Time

For the computation time experiments we have used the SemCor files and performed disambiguation for the 47 words used in experiment 2. The average time by word disambiguation is shown in table 5.27. These results were achieved in an INTEL Celeron at 1.0 GHz, with 512Mb of RAM, running Windows 2000. There are huge differences in these figures. While simple methods are very fast, like Entry Sense

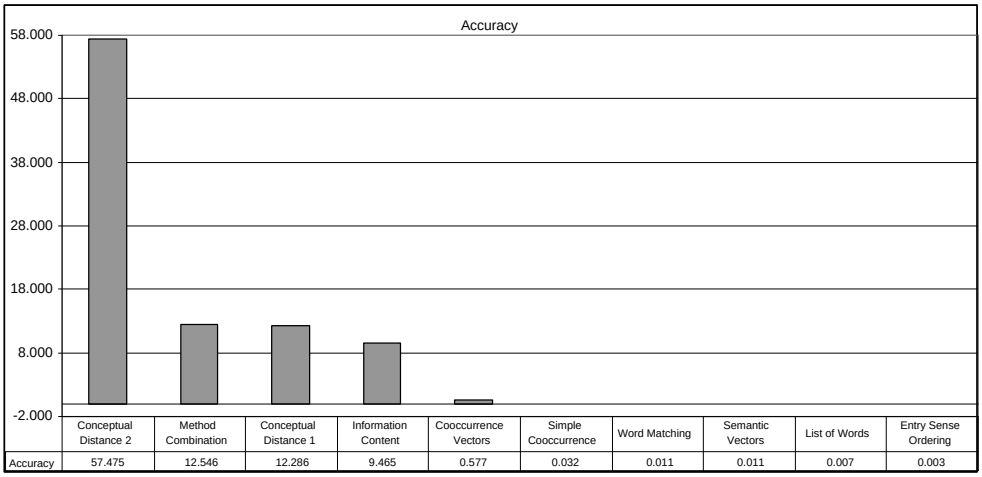


Figure 5.27: Experimental results obtained for the computation time with the SemCor 1.7 disambiguation.

Ordering or Word Matching, complex methods using semantic distances computed using WordNet are time consuming, due to the high average number of synsets by word.

Chapter 6

Related Work

This chapter describes several systems and works related to REBUILDER, comparing them with our approach. We selected works that are somehow related to our approach, concerning the goal and application domain of the system, or systems which deal with some specific problems addressed in REBUILDER.

REBUILDER is a design tool integrating several reasoning techniques and knowledge types. Due to the numerous aspects that are handled in REBUILDER, there are several related systems and approaches that need to be compared with our system. These can be categorized in several classes: CBR design systems, CBR software design systems, analogy software design systems, other software reuse systems, systems dealing with analogical retrieval, systems that use design patterns, and works on word sense disambiguation.

Concerning the systems more similar to REBUILDER, we point out two types of software reuse systems: CBR based and analogy based. These two types of systems have some points in common, especially in which concerns to the reasoning paradigm. We compare them along four application axis:

- Phase of the software development process the system is intended to address (analysis, design or implementation);
- Programming paradigm that is supported (procedural, functional, object-oriented or declarative);
- Level of abstraction of the domain of application (generic or specific);
- Incorporation of adaptation or just retrieval.

In the following sections, each paragraph describes a different system, stressing it's main characteristics, along with some relevant differences concerning to REBUILDER.

6.1 CBR Design Systems

This section describes several research works on design systems using CBR. Some systems are not in the software design or software reuse domain, but have important common aspects with REBUILDER. These systems represent landmarks in the CBR design systems, and so it is important to compare them with our approach.

EADOCS [Netten and Vingerhoeds, 1996] is a support system for conceptual design of thin-walled fibre reinforced by composite panels in aircraft structures. It uses a multi-level approach to design generation, applying CBR, Rule-Based Reasoning (RBR) and Constraint-Based Reasoning (CtBR). Cases are represented by an hierarchical structure of objects defined as attribute/value pairs, and they are treated as prototype cases capturing abstract knowledge. Cases are not indexed and are stored in a simple list. Case retrieval and ranking are performed at the functional level. Conceptual design in EADOCS comprises three different phases: generation of qualitative solutions, designated as prototypes; instantiation of the generated prototype through parameter instantiation; and quantitative adaptation and optimization. CtBR is used in prototype retrieval, while CBR is used for the retrieval of a case that is going to be used in the prototype quantification. Adaptation and optimization are performed using RBR and domain specific heuristics. EADOCS is an encouraging example of multiple reasoning integration.

COMPOSER [Purvis and Pu, 1995, Pu and Purvis, 1997, Purvis and Pu, 1998] is a system for design of assembly sequences, which is the problem of finding a valid sequence to assemble several parts into a final product. The approach of COMPOSER is based on CBR and CtBR. Cases are described by attribute/value pairs and have constraints associated to them. Retrieval is based on structural correspondences, and adaptation is performed using a constraint satisfaction algorithm. The combination of these two reasoning mechanisms enables the system to start with a draft of a first solution, which is then improved into a final solution using CtBR. COMPOSER is a

design system that works at the configuration level, using only structural knowledge. REBUILDER is able to work with other design knowledge levels, like functional and behavioral. A similarity with REBUILDER is that the adaptation algorithm is able to combine parts of different cases into the same solution. In COMPOSER this is performed using the constraint satisfaction algorithm, while in REBUILDER this is achieved using design composition.

CADET [Sycara and Navinchandra, 1991, Sycara and Navinchandra, 1993] is a system that uses behavior as a thematic abstraction for case retrieval. The application domain is mechanical design. It uses influences as a thematic abstraction for representing behavior of physical devices. In this framework, case organization provides the main mechanism for cross-contextual reminding which is very important in non-routine design. In CADET, a case is described by two parts: solution and behavior. The case behavior is represented by qualitative influences, which are qualitative differential (partial or total) relations between two variables. One of which is a dependent variable and the other is an independent variable. These influences are represented by graphs describing the behavior of structural components. Influences index cases using a hierarchical thematic abstraction structure. They are also used for retrieval and other reasoning tasks. More specifically, retrieval is performed through the matching of the query influence graph with cases in the library. Solution verification and evaluation is performed using qualitative and quantitative simulations of influences. CADET works at two levels of design knowledge: behavioral and structural.

CADRE [Dave et al., 1991] is a system for solving design problems concerning spatial arrangement in buildings. This system addresses four main issues: case adaptation, case composition, integration of several different levels of knowledge abstraction, and interactive support for the designer. It uses CBR, with cases representing structural components using a spatial representation. Cases are represented from two different user perspectives: architect and engineer. It has no indexing structure able to speed up the retrieval process. Case adaptation is based on CtBR, and is performed in two distinct steps: dimensional reduction of design constraints, and if this is not enough to solve the problem, then it can modify the design topology. Design composition is based on the design topology and constraints. Retrieval is performed using the topological properties of cases, and selected cases are ranked using

structural graph matching.

Smith developed IDIOM [Smith et al., 1995], which is a follow up of CADRE. IDIOM introduces several improvements over its antecessor, particularly in the adaptation process, where models Function-Behavior-Structure (FBS) are used to guide the adaptation process. At the user interface, it uses the concept of intelligent object. An intelligent object uses three different types of knowledge: specific knowledge from a case, designer interpretation of the object, and the object's model. Designer interaction is based on these objects, which enables the dynamic definition of the target problem.

FABEL [Voss et al., 1994, Coulon and Gebhardt, 1994, Gebhardt et al., 1997] is a case-based design system in the architectural domain. It is intended to function as an intelligent CAD tool, providing a high level support to architects and engineers, based on several tools. These tools cover several cognitive support functions like: case retrieval, design analysis, object classification, design adaptation, design generation, and team development management. FABEL uses not only cases, but also models and prototypes. Cases have a complex representation, combining object-based with graph-based representations. Though cases are represented only in structural terms, behavioral knowledge about a case can be obtained from domain models, while functional knowledge can be obtained from prototypes. The target problem is defined as an incomplete structure that the system has to complete. Case retrieval comprises several methods, which are used to help the designer to retrieve and explore different aspects of the design. Several indexing structures are used in FABEL: associative memories, vectors of attribute/value pairs, decision trees, and concept taxonomies. One aspect of FABEL, similar to REBUILDER, is that it uses a visual editor to interact with the designer. In this way, both systems represent cases as diagrams, FABEL in the architecture domain, and REBUILDER in the software design domain. FABEL uses mainly rule-based adaptation [Voss, 1997], but is capable of using other adaptation methods such as constraint satisfaction or generate and test. Most of these methods involve domain knowledge, which might not be available in some domains. In software design, part of the world is modelled as a software system. So, the domain knowledge is the domain of application, which can be anything. This makes adaptation limited to the domains for which the system was developed. In REBUILDER, the goal is to build a generalist system that is not dependable of the

type of software system.

Voss developed TOPO [Voss and Coulon, 1996] which is a design system in the architectural domain, and part of the FABEL project [Gebhardt et al., 1997]. We have chosen to present TOPO apart from FABEL due to its particular characteristics, since it can be considered a system within a system. The main goal of TOPO consists on supporting an architect during the design process. This system characterizes itself for its high level of interaction with the designer. TOPO uses CBR and represents cases by it's structure. There is no functional or behavioral description. The target problem is provided to TOPO in terms of structural graphs, which the system then completes using CBR, a process similar to REBUILDER. Cases are retrieved from the case library using structural indexes and retrieved cases are ranked using the structural similarity in the form of graph matching. Adaptation is made through structural operations, deleting, adding or replacing structural components. TOPO uses some heuristics to assess the adaptation effort needed to adapt a case, which can then be used to abandon adaptation and select another retrieved case.

Mary Lou Maher has developed several case-based design systems (CASECAD, CADSYN, WIN, DEMEX) [Maher et al., 1995, Maher, 1996] in the architectural domain. The first system to be developed was CASECAD, followed by CADSYN which introduces other reasoning mechanisms. WIN was the next system and focus in a more specific issue: designing buildings taking into concern wind resistance. The last system of this 'family' is DEMEX that tries to provide the designer with support in the exploration of new design space regions. REBUILDER shares some of the features of these systems, namely the ability to explore the memory of previous designs (the case library). The next paragraphs describe each of these systems in more detail.

CASECAD is a design support system in the domain of architecture based on the integration of CBR and CAD technology. It has two main working modes: browsing and retrieving. It allows the designer to explore the design space by browsing the case base, and it can also retrieve designs using a query in the form of attribute/value pairs. It's indexing structure is divided into models and cases. The models are organized into an hierarchical structure that enables case indexing by classification. Reasoning mechanisms can take advantage of domain knowledge in the form of models. Case representation includes several formalisms: multimedia information, object-oriented

representation, CAD drawings and graph illustrations of case behavior. Models can also be used for problem elaboration, making it more complete and specific, or they can be used to guide adaptation. Cases can be indexed by subparts like in REBUILDER, this enables flexible case retrieval and composition of different case pieces into a new solution. This indexing is only performed at the structural level. More specifically, cases are indexed by classification and by attributes. Case retrieval is performed in two steps: first the algorithm retrieves cases with the same problem classification; then problem and case attributes are matched. At the end, the system reviews all the cases with the same attributes and classification. Ranking is based on the number of common attribute/value pairs. Cases can be indexed by any type of attribute, be it structural, behavioral or functional. This happens because a target problem can be specified using any type of attribute. Adaptation is left to the designer. Nevertheless models can be used to aid this process.

CADSYN is very similar to CASECAD, but it can also search for subsystems using a hierarchy of requirements. CADSYN adds constraint satisfaction algorithms for adaptation to CASECAD, and does not give great attention to the graphical interaction with the user. CADSYN indexes cases using functional indexes, and ranks retrieved cases based on the number of common functionalities with the target problem. Retrieved cases are the ones that have at least one common functionality with the problem specification. Case adaptation is based on a constraint satisfaction algorithm. If the adaptation process is unable to generate a new design, then the problem is decomposed in subproblems and each one of these subproblems is solved independently. The system then gathers all the partial solutions and composes a new solution.

WIN is a CBR design support system with an emphasis on the structural perspective of civil engineering. The application domain is the same as CASECAD and CADSYN, but centered in more specific aspects, related with the wind resistance of high buildings. In WIN, case retrieval is an iterative and interactive process that uses model-based reasoning (MBR) retrieval besides doing the attribute/value pair case retrieval. As in its predecessors, the knowledge base comprises both models and cases. Cases are represented by vectors of attribute/value pairs. Attributes can belong to four categories: relation, function, behavior and structure. Models, as well as cases, can be decomposed in sub models. Case retrieval is performed in an interactive

way. First the designer provides a problem description based on attribute/value pairs, which are used as indexes to retrieve a first set of relevant models. These models are used to elaborate and complete the problem description, which enables a more accurate case retrieval. After the cases are retrieved, the system presents them to the designer. As CASECAD, WIN does not perform automatic adaptation.

Finally DEMEX addresses the issues of flexible retrieval and memory exploration using three different types of memory: associative memory, model memory and case memory. DEMEX addresses the same application domain as its predecessors. It has the same case representation as CASECAD. A major difference between DEMEX and previous systems is its indexing structure. It comprises three different parts: an associative memory that relates functional, behavior and structural categories to design attributes; a model structure; and a case structure. Each case is generalized in the form of a model. Each model indexes a set of cases that are its specializations. In this way is it is possible to navigate the indexing structures. The retrieval process consists on following these structures and look for the attributes that are common with the target problem specification. Then it follows through the links that represent the attribute values, corresponding to the problem, reaching appropriate models, which are then used for problem elaboration. Finally, it reaches a set of cases that are retrieved. DEMEX does not perform automatic adaptation, leaving it to the designer.

The work developed by Ashok Goel with the KRITIK family of design systems [Goel, 1991, Goel, 1992, Bhatta and Goel, 1993a, Bhatta and Goel, 1993b, Goel et al., 1997a, Goel et al., 1997b, Bhatta and Goel, 1997a, Bhatta and Goel, 1997b] combine CBR with MBR. In his work, Goel uses a case representation that describes a design at three different levels: function, behavior, and structure (SBF representation formalism[1]), which allows cases to be indexed and retrieved using these three design aspects. REBUILDER uses only functional indexes (object synsets), but it provides a functional taxonomy for the indexing mechanism allowing the computation of similarity distances between these indexes. The KRITIK family comprises four systems (by chronological order): KRITIK, IDEAL, KRITIK2 and INTERACTIVE KRITIK. All these systems have the same application domains: mechanical and electrical system design.

[1]These models are based on the component-substance ontology developed by Bylander and Chandrasekaran [Bylander and Chandrasekaran, 1985].

The first system to be developed was KRITIK, which was the base for all the following systems. It uses SBF models to represent cases, allowing indexing, retrieval, adaptation and verification of design cases. Problem specification in KRITIK is performed using functional knowledge. So, cases are indexed using functional indexes. The system uses a list of functions as indexing structure. In this list, each element is a functionality that indexes one or more cases. Retrieval is performed by traversing this list looking for cases with functionalities common to the target problem. After identifying these cases, they are ranked based on the number of common functionalities with the problem specification. After selecting the best case, KRITIK adapts the solution of the retrieved case to the problem. SBF models are used in this task to compute the functional differences between the retrieved case and the problem. Using SBF models, functional differences correspond to changing behaviors, and behavioral changes correspond to structural adaptations. KRITIK uses adaptation plans to eliminate these functional differences. Verification and evaluation of the new design is made using qualitative simulation of the SBF model associated with the design.

IDEAL is the successor of KRITIK, and presents two major differences. The first one, is that, it is able to work with generic behavior models (BF models), besides normal SBF models. These models are the result of a learning process consisting in the generalization of cases (represented as SBF models). BF models are then used for adaptation purposes, using an analogy mechanism. The second important difference is the case indexing. IDEAL uses functional indexes as KRITIK, but it adds a second type of indexes: structural indexes. These indexes are selected based on BF models, which are able to identify important structural components in a device behavior. This improvement enables problem specification in terms of structural constraints in addition to functional requirements. Goel and Bhatta developed model-based analogy, a computational theory of analogy-based design that uses domain models about artifacts. These models represent the structure, behavior and function of a design. It uses design patterns[2] as generic design abstractions that are learned during system operation. A design pattern can then be applied to a concrete situation, generating a new analog design.

KRITIK2 was the successor of IDEAL resulting from the lessons learned with

[2]The design pattern concept used here, is different from the one used in software design. For Goel a design pattern is a general template for a SBF model.

the previous systems. The analogical reasoning mechanism is improved, such that the system is able to make cross-domain analogies. This enables the emergence of creative design properties in KRITIK2. The system uses also new types of indexes, which result from the design outcome. These indexes represent the outcome of the design, describing if the design was successful or not and which were the failure elements.

INTERACTIVE KRITIK integrates an user interface in the system, providing a graphical interpretation of SBF models, which is very useful for the designer.

IM-RECIDE [Bento, 1996, Gomes et al., 1996, Gomes et al., 1998] was developed in the University of Coimbra, and was a first and far predecessor of REBUILDER. IM-RECIDE is an expert system development shell based on CBR. This system focus on two major research issues: reasoning with cases imperfectly described and explained, and the exploration of the case library in order to generate creative solutions. A case comprises three different parts: problem description, solution description and explanation. Problem and solution are described by attribute/value pairs, while explanations are represented by causal trees, connecting problem facts to solution facts. This is a representation tailored for diagnosis and classification domains, which shows to be semantically poor for design purposes. Case retrieval is based on partitions of the case base. These partitions are called spaces (also called design spaces). The definition of each space is based on the problem characteristics (each problem defines different partitions in the case base). Four spaces are defined: space I, space II, space III, and space IV. Space I comprises the cases similar to the target problem. From space I to space IV cases are gradually more distant from the problem specifications. IM-RECIDE tries to solve a problem going from space I to space IV, using a similarity metric [Bento et al., 1995] within each space, and a set of specific adaptation operators that generate new designs in each space. IM-RECIDE has no indexing structure, which explains some of its poor performance figures. The systems performs verification and evaluation of new designs, along with the learning of new cases.

CREATOR [Gomes and Bento, 1997b, Gomes and Bento, 1998] is a case-based reasoning shell for creative design. In the problem specification phase a graphical editor is used to help the user identifying the desired design functionalities. CREATOR

uses its memory structure to perform problem elaboration in order to complete the design requirements [Gomes and Bento, 1997a]. For problem elaboration it uses two types of knowledge represented in the memory structure: a hierarchy of functionalities and the problem descriptions for the cases in memory. Design episodes are represented in the form of SBF models [Goel, 1992]. A case comprises three different parts: (1) problem specification - function; (2) explanation - behavior; (3) design solution - structure. Case retrieval is performed in an incremental way, using the indexing structures, which are based on a functional taxonomy. CREATOR has a retrieval algorithm sensitive to the adaptation mechanism intended to be used. Adaptation uses three operators: mutation, composition, and thematic abstraction. Ranking is based on two criteria: the distance between the target problem and the case classification, and the differences between the target problem and the retrieved case. CREATOR does not perform adaptation or verification, which are left to the designer.

Tables 6.1 and 6.2 present a summary of the characteristics of systems presented in this section. All these systems were developed for a different design domain when compared with REBUILDER, though there is a common ground, which is the design task. Most of the systems use MBR, besides CBR. One possible explanation is the representation formalism that is used, which enables this type of reasoning. One main difference in this aspect between REBUILDER and the other systems, is the use of analogy and natural language processing, providing the system with the capacity to generate new solutions potentially creative, and to use natural language as a way of communication with the designer.

Most of the design systems presented use attribute/value pairs or models to represent designs. Attribute/value pairs have the advantage of being simple to manipulate, but they have some limitations concerning expressiveness. Models are powerful instruments to represent designs, and enable MBR which is important for tasks such as adaptation and verification. We think that these two representation formalisms, although they may be adequate to represent software designs, have a limitation when compared with UML class diagrams - they are not a natural way for communication with the designer. UML has the advantage of being the "natural language" for software designers. Besides this main difference, UML is easier for semantic interpretation than, for instance, SBF models. Which in our opinion are more difficult to apply to software designs.

Table 6.1: Comparison table for the main characteristics of CBR design systems, part (a). PBR stands for Prototype-Based Reasoning.

Name	Application Domain	Reasoning	Case Representation	Memory Structure
EADOCS	Aircraft Structures	CBR, MBR and RBR	Attribute/value pairs	None
COMPOSER	Assembly sequence	CBR and CtBR	Attribute/value pairs	None
CADET	Mechanical Design	CBR and Qualitative Reasoning	Attribute/value pairs and Influences	Hierarchical Structure
TOPO	Architecture	CBR	Attribute/value pairs	None
CADRE	Architecture	CBR and CtBR	Attribute/value pairs	Hierarchical structure
IDIOM	Architecture	CBR, MBR and CtBR	Attribute/value pairs	Hierarchical structure
FABEL	Architecture	CBR, MBR and PBR	Attribute/value pairs	Associative and hierarchical
CASECAD	Architecture	CBR and MBR	Attribute/value pairs, models and CAD drawings	Model hierarchy
CADSYN	Architecture	CBR, MBR and CtBR	Attribute/value pairs and models	Model hierarchy
WIN	Architecture	CBR and MBR	Attribute/value pairs and models	Model hierarchy
DEMEX	Architecture	CBR and MBR	Attribute/value pairs, models and CAD drawings	Hierarchy of models and associative memory
KRITIK	Mechanical and electrical design	CBR and MBR	SBF models	Function list
IDEAL	Mechanical and electrical design	CBR and MBR	SBF models	Function and structure list
KRITIK2	Mechanical and electrical design	CBR and MBR	SBF models	Function, structure and outcome lists
INTERACTIVE KRITIK	Mechanical and electrical design	CBR and MBR	SBF models	Function, structure and outcome lists
IM-RECIDE	Generic	CBR	Attribute/value pairs and causal explanations	None
CREATOR	Generic	CBR and MBR	SBF models	Function hierarchy and associative memory
REBUILDER	Software design	CBR, Analogy and NLP	Graph-Based	WordNet

Table 6.2: Comparison table for the main characteristics of CBR design systems, part (b).

Name	Indexing	Retrieval	Adaptation	Verification
EADOCS	Function-based	Function similarity	CtBR	None
COMPOSER	Structure-based	Structural similarity	CtBR	None
CADET	Behavior-based	Similarity of the behavior graph	Rule-based	Qualitative and quantitative simulation
TOPO	Structure-based	Similarity of the structural graph	Rule-based	None
CADRE	Structure-based	Similarity of the structural graph	CtBR	None
IDIOM	Structure-based	Similarity of the structural graph	CtBR	None
FABEL	Structure-based	Several algorithms	MBR, RBR and PBR	RBR and MBR
CASECAD	Structure, behavior and function-based	Function similarity	None	None
CADSYN	Structure, behavior and function-based	Function similarity	CtBR	None
WIN	Structure, behavior and function-based	Function similarity	None	None
DEMEX	Structure, behavior and function-based	Function similarity	None	None
KRITIK	Function-based	Function similarity	MBR	MBR
IDEAL	Function-based	Functional and structural similarity	MBR	MBR
KRITIK2	Function and structural and outcome-based	Functional, structural and outcome similarity	MBR	MBR
INTERACTIVE KRITIK	Mechanical and electrical design	Functional, structural and outcome similarity	MBR	MBR
IM-RECIDE	None	Functional similarity	RBR	Failure cases
CREATOR	Function-based	Functional similarity	None	None
REBUILDER	Function-based	Function, behavior and structural similarity	Analogy, CBR and Design Patterns	Verification and design cases, and WordNet

The memory structure (or indexing structure) is crucial for fast and efficient retrieval. Most of the systems have a hierarchical structure that organizes indexes, allowing retrieval to perform search in this structure. Most of the systems use a functional taxonomy or a model hierarchy, depending on their type of case representation and query formulation. REBUILDER uses WordNet as a functional taxonomy, allowing the use of synsets for case indexing. Most of the systems index cases by function, but there are also some systems that additionally use structure and behavior. REBUILDER performs only function-based indexing, because retrieval probes are object synsets. Retrieval is restrained to the indexes used by the system. REBUILDER and most of the other systems use function-based retrieval. Then REBUILDER applies a similarity metric that also uses structure and behavior to rank retrieved cases. Some other systems also use this two-phase retrieval, but there are some systems that are limited to function similarity for ranking the retrieved cases.

Most of the systems use only one adaptation method, the exception is FABEL that uses MBR, RBR and PBR. REBUILDER has an elaborated approach to adaptation comprising three types of adaptation mechanisms. The most used methods of adaptation are MBR, CtBR and RBR. Verification is another aspect that we tried to cover extensively in REBUILDER, using most of the knowledge sources at our disposal. Approximately half of the systems presented in this section do not address verification, and the ones that do so use MBR that is enabled by it's case representation formalisms.

6.2 CBR Software Design Systems

González et. al. [Fernández-Chamizo et al., 1996, González and Fernández, 1997, González-Calero et al., 1999] presented a CBR approach for reuse of object-oriented code. They combine description logics with CBR for reasoning purposes. Cases represent three types of entities: classes, methods and programming recipes, allowing the retrieval of these types of objects. Cases comprise a problem (lexical description), a solution (code) and a justification (code properties). They use LOOM, a conceptual description language, for representation of code properties. The system uses a lexical

retrieval algorithm starting from a natural language query. It also performs conceptual retrieval using an entity and slot similarity measures. One difference between REBUILDER and this approach, is that our system works at the design level, while González's approach deals with the coding level. Although this system also performs design reuse, it's main focus is on code reuse. Design reuse is as a byproduct of code reuse, when classes and interfaces get moved or reused as blocks. Besides this difference, González's work is based on a specific description language, while our approach is based on UML, which is a standard and widely used software design language. The system allows lexical or conceptual retrieval using LOOM. One particularity of this approach is that cases comprise justifications, allowing the explanation of solutions. Case reuse is based on substitution-based adaptation, which uses the dependencies within a case to guide the process. The system learns successful search episodes as search heuristics, in order to improve retrieval performance.

Déjà Vu [Smyth and Cunningham, 1992, Smyth and Keane, 1995a, Smyth, 1996] is a CBR system for code generation and reuse using hierarchical CBR. It reuses code for control of autonomous vehicles in loading and unloading cargoes. Code is in a procedural form. Déjà Vu uses a hierarchical case representation, indexing cases by functional features. The main improvement of this system is the adaptation-guided retrieval mechanism that retrieves cases based on the case adaptation effort instead of the similarity with the target problem. Déjà Vu is based on hierarchical CBR with a blackboard control structure. Retrieval and adaptation specialists are used for reasoning and are controlled by the blackboard system. This work has several points in common with REBUILDER. Some of these similarities are the hierarchical case representation and reasoning, and the functional indexing of cases. Despite these similarities, Déjà Vu deals with the coding phase of software development and not with the design phase. The indexing structure that is used is a taxonomy of functional classification nodes specific to plant-model planning. Each classification node is represented by frames. Adaptation is based on rules and it involves interaction between the designer and the system.

Krampe and Lusti [Krampe and Lusti, 1997] developed an approach for reuse of design specifications in information systems. They generate new specifications using design composition as a form of adaptation. They combine three different reasoning paradigms: CBR, MBR and schema-based reasoning. CBR is used for representation

and reuse of design specifications. MBR is used to check design consistency through the use of meta-models. Schema-based reasoning is a reasoning paradigm, which is not as specific as CBR, and not as abstract as MBR. It is something between the two. Schema-based reasoning uses reference models to perform inferences. Cases comprise specifications and components. Specifications represent the situation or problem described in the case. Components represent the resulting design (the case solution). The indexing structure comprises several meta-models, which are organized in the form of classification hierarchies. Cases are indexed by their components, which are classified by the function that they perform. Retrieval is based on specifications and components of the target problem. Adaptation consists on two steps: internal adaptation by the system and external adaptation by the designer. The system then uses integrity rules and/or reference models to verify and evaluate the generated design. All the new cases are added to the case library, without a case base maintenance strategy.

CAESER [Fouqué and Matwin, 1993] is another CBR tool for code reuse. It works at the code level and uses data-flow analysis to acquire functional indexes. The user can retrieve cases from the case library using a prolog-like query, which is used by the system to retrieve similar functions. A case comprises a problem description and a solution. The problem is represented through the use of frames that describe the case functionalities. The solution is the code. It enables reuse at the code level and it is capable of performing adaptation, through the use of knowledge-based rules for composition. It also allows the testing and evaluation of generated functions using cause-effect graphs. It is based on control and data-flow analysis to make structural and functional decomposition. CAESER was developed for procedural code (mathematical functions), and does not deal with object-oriented design. It reuses procedural code organized in functions or procedures.

Althoff and Tautz [Althoff et al., 1997, Tautz and Althoff, 1997] have a different approach to software reuse and design. Instead of reusing code, they reuse system requirements and knowledge on software development. This approach is one of the few approaches that address all the phases of software development. They reuse system requirements and associated knowledge on software development in the form of electronic documents. This approach is very different from previous ones and also from our approach, because it deals with document retrieval and indexing, thus having a

knowledge management flavor. One important limitation of Althoff's approach comes from the lack of reuse or revise phases. Instead of reusing code or design specifications through the understanding of its components, or using a specific representation language, they reuse any type of software development document or code using a superficial representation based on vectors of attribute/value pairs. This approach is intended to be applicable to any type of software. Adaptation is very limited and code/design understanding is not addressed in the developed methodology.

CREATOR II [Gomes and Bento, 1999b, Gomes and Bento, 2001] is a CBR system that helps the programmer developing VHDL code through the reuse of previous VHDL functions. Cases are represented in Function-Behavior Case Representation (FBCR). This formalism is designed for description of procedural software languages, ranging from general purposed languages like C, to more specific languages, as VHDL. FBCR is derived from the SBF models developed by Goel [Goel, 1992]. Because software programs can be seen as designs, FBCR describes a software program at the functional and behavior levels. CREATOR II comprises four modules: VHDL to FBCR converter, FBCR to VHDL converter, Case-Based Reasoning module, and Knowledge Base. The VHDL to FBCR converter converts a VHDL file into a FBCR case [Gomes and Bento, 1999a]. In this process functional and behavioural knowledge is extracted from the VHDL code and used for indexing. The functional and behavioural knowledge is also used for case adaptation and verification. The knowledge base comprises four parts: case library, indexing structure, function taxonomy, and data taxonomy. The indexing structure is used for problem elaboration and retrieval of cases from the case library. The system uses a function taxonomy as domain knowledge. It comprises a hierarchy of functions in the domain of digital circuit design. The data taxonomy is a hierarchy of data types used in the VHDL language and it also represents the domain knowledge used by the system for reasoning tasks. This system is the closest predecessor to REBUILDER.

Systems presented in this section have more in common with REBUILDER than the systems in the previous section, because they are CBR systems for the domain of software reuse. A summary of the characteristics of these systems is presented in tables 6.3 and 6.4. Notice that four out of six systems only reuse code, and only one is tailored for reuse of object-oriented code. The others deal with procedural code. This makes a strong difference against our work. Only Althoff's system deals with

Table 6.3: Comparison table for the main characteristics of CBR software design systems, part (a).

Name	Application Domain	Reasoning	Case Representation	Memory Structure
González's	Software Reuse [Code, Object-Oriented]	CBR and description logics	attribute/value pairs, code and lexical	None
Déjà Vu	Software Reuse [Code, Procedural]	CBR and blackboard systems	Hierarchical representation	Taxonomy of classification nodes
Krampe's	Reuse of design specifications [Analysis, All paradigms]	CBR, MBR and Schema-based reasoning	attribute/value pairs	Meta-models arranged in hierarchies
CAESER	Reuse of mathematical functions [Code, Procedural]	CBR and data-flow analysis	Frames (problem) and code (solution)	Functional taxonomies
Althoff's	Reuse of software development knowledge [All, All]	CBR	Attribute/value pairs	None
CREATOR II	Reuse of VHDL software [Code, Procedural]	CBR and MBR	FBCR and code	Functional taxonomy
REBUILDER	Reuse and design of Software [Design, Object-Oriented]	CBR, Analogy and NLP	Graph-Based	WordNet

the design phase, but at a more abstract level than REBUILDER. In another way this work deals with all development phases.

In terms of reasoning mechanisms, all the systems described use CBR and another reasoning mechanism. This secondary reasoning mechanism is used to complement the CBR reasoning in tasks such as retrieval, adaptation or verification. Case representations range from attribute/value pairs (used by the systems that deal with all the programming paradigms, which makes sense, since there is a need for a more general and simple case representation) to more complex representations like the model-based representation or hierarchical representation. These complex representation formalisms are suitable for more specific and restrict domains of application, which is the case of these systems and also of REBUILDER. An intermediate representation is the frame-based representation used by CAESER, which comprises attribute/value pairs organized in a structure. REBUILDER uses a graph-based representation, in the form of class diagrams. Attribute/value pairs provide a representation simple for reasoning, but they have less expressive power comparatively to class diagrams. The main advantage of class diagrams is that the internal and external representation of designs is the same.

Memory structure and indexing in the systems presented here is very alike, with most of the systems using function-based indexing with a taxonomy of functions as

Table 6.4: Comparison table for the main characteristics of CBR software design systems, part (b).

Name	Indexing	Retrieval	Adaptation	Verification
González's	lexical-based	Using description logics and conceptual retrieval	Substitution-based	None
Déjà	Function-based	Functional similarity and adaptation-guided	Rule-based with user interaction	None
Krampe's	Function-based	Functional similarity	Heuristics	Integrity rules and reference model rules
CAESER	Functional-based	Functional and behavioral retrieval	Knowledge-based rules for composition	Testing code using cause-effect graphs
Althoff's	None	Attribute similarity (KNN)	Adaptation rules	None
CREATOR II	Function-based	Functional similarity	None	None
REBUILDER	Function-based	Function, behavior and structural similarity	Analogy, CBR and Design Patterns	Verification and design cases, and WordNet

the index structure. This is not surprisingly, since the design problem is specified in functional terms, and hierarchies are one of the most efficient ways to organize indexes. González's system uses lexical-based indexing and does not have a memory structure. Other systems do not have indexes or indexing structure. Retrieval in almost every system is performed using the functional indexes and the function taxonomy. Systems like Déjà Vu, complement this with adaptation-guided retrieval, or behavior similarity. REBUILDER goes further in this way and uses functional, behavioral and structural similarity to rank retrieved cases. In Althoff's work, and due to the case representation approach, he uses K-Nearest Neighbor to retrieve and rank cases. González's work is slightly different and uses description logics and conceptual retrieval, which can be viewed as a type of functional retrieval.

Four of the systems presented (Déjà Vu, Krampe's system, CAESER and Althoff's system) have some kind of adaptation rules or heuristics, providing a limited adaptation mechanism. González's system uses substitution-based adaptation that is possible due to the description logics that is used. REBUILDER explores a richer and different set of mechanisms, like analogy, design composition and design patterns. While rules and heuristics used in other systems are domain dependent, the mechanisms used in REBUILDER are domain independent, except the Design Patterns mechanism. In the verification step, only two systems (Krampe's system and

CEASER) perform verification of new solutions. Krampe's performs verification using integrity rules and models, while CAESER uses cause-effect graphs, which can be interpreted as models. REBUILDER uses a different approach performing design verification supported on three types of structures: verification cases, design cases and WordNet. While models, rules and cases are domain specific, WordNet is a generic approach to verification.

6.3 Analogy Software Design Systems

Maiden and Sutcliffe [Maiden and Sutcliffe, 1992] developed an approach to software specification reuse based on analogy. They focus on reasoning with critical problem features, retrieving analogous specifications of the same category. A domain abstraction is used as an intermediate domain in the mapping between the target and the source. Their system, Ira, uses analogical mapping and transfer of knowledge. Ira works with object oriented specifications and it does not have a specific domain of application. The reuse of specifications involves three steps: categorization of a new problem, selection of candidate specifications, and adaptation of the selected specification to the new domain. Ira addresses these three issues by: obtaining the description of the new target problem from the software engineer; controlling the interaction with the user during selection and adaptation of an analogous specification; and reasoning with critical problem features to match new problems. The problem specification is described using predicate logic, and it uses abstract domain models as knowledge for the process. The retrieval of analogue candidates is an iterative process based on component categorization. The system uses heuristics to compute the degree of difference between two abstract domains, and also to identify analogical matches. The analogical engine maps semantically equivalent predicates representing critical structures.

Spanoudakis [Spanoudakis and Constantopoulos, 1994] developed a computational model of similarity for analogical software reuse based on conceptual descriptions of software artifacts, intended to be of generic application. Their approach is based on semantic similarity of software objects. They developed TELOS a conceptual modelling language used to represent object oriented code and design. TELOS

enables the use of analogical software reuse.

Jeng and Cheng [Jeng and Cheng, 1993] present an approach for code reuse based on analogy and formal methods. They argue that in order to use analogy, code representation must be defined in a formal way, which can be provided by formal methods. They reuse functional code and the application domain is generic. Code is described using a formal specification. Retrieval of analogue candidates is based on a top-down matching process based on analogies between pairs of components.

Harandi and Bhansali [Harandi and Bhansali, 1998] also presented an approach to derive programs based on analogy. Their system is based on heuristics to guide candidate selection, followed by derivational transformation methods for adaptation of the selected candidates. The application domain selected is the development of C code for the Unix operating system. A semi formal representation is used to specify the target problem. The derivational transformation comprises two steps: decomposition of the problem in subproblems; and then use of several application rules. They consider three rule levels: high level strategies for problem solving, which are generic; rules for solving problems in the operating system domain; and rules specific to the Unix operating system.

ROSA [Tessem et al., 1994] reuses code and design specifications with a modelling language named OOram. This language provides the representation formalism necessary to perform analogy on object oriented code in any type of domain. ROSA is integrated in a CASE tool using the OOram modelling language. ROSA uses WordNet with the role of associating synsets to model objects, but it does not perform word sense disambiguation automatically. The designer must select the correct sense for the model element. Components are indexed along a set of dimensions, which are defined by several heuristics from analogy studies. This system can store successful analogies for later use. To be stored, new analogies must have a score higher than a predefined threshold value. Despite associating synsets to objects, ROSA does not use the synsets as indexes for retrieval.

Table 6.5 provides an overview of the main characteristics of the systems presented in this section. Most of these systems deal with code reuse, the exceptions are Maiden's work that addresses system analysis, ROSA that provides support for design reuse, and REBUILDER. In ROSA design reuse is a byproduct of code reuse,

Table 6.5: Comparison table for the main characteristics of analogy software reuse systems.

Name	Application Domain	Knowledge Representation	Candidate Selection
Maiden & Sutcliffe	Specification Reuse [Analysis, Object-Oriented]	Predicates and abstract domain models	Semantic-based
Spanoudakis	Code Reuse [Code, Object-Oriented]	TELOS conceptual modelling language	Semantic and structure-based
Jeng & Cheng	Code Reuse [Code, Procedural]	Formal specifications	Semantic and Structure-based
Harandi & Bansali	Program derivation [Code, Procedural]	Semiformal specification	Semantic and Structure-based
Tessem (ROSA)	Code and Design Reuse [Code+Design, Object-Oriented]	OOram language	Semantic-based
REBUILDER	UML diagram reuse [Design, Object-Oriented]	Graph-based (UML class diagrams)	Function, behavior and structure-based

which is the main goal of the system. The knowledge representation formalisms that are used vary from system to system. Most systems use a specific representation language to describe designs or code. The only system that uses a widespread representation language is REBUILDER with UML. For candidate selection, all systems use at least semantic similarity. The majority of these systems use also structural similarity, in the retrieval algorithm or in the similarity metric used to rank candidates. Besides semantic and structural similarity, REBUILDER takes into account behavior similarity in the metric used to rank candidate cases.

6.4 Other Software Reuse Systems

The earlier code reuse systems were based on a library of procedures or objects, which could be retrieved using a query system. The work developed by Prieto-Díaz [Prieto-Diaz and Freeman, 1987, Prieto-Diaz, 1991] is representative of these initial systems. The main difference of this approach, against the idea behind REBUILDER, is that our system focus on design reuse. The main automated functionalities in this system were retrieval and retain, while REBUILDER adds reuse and revise capabilities. Prieto's approach to code reuse is based on a faceted classification of software components. According to Prieto-Díaz, faceted classification has two different aspects. It must classify components by the functions it performs and by the environment in

which it works. Conceptual graphs are used to organize facets, and a conceptual closeness measure is used to compute similarity between facets. He has identified six facets for software components: function, object, medium, system type, functional area and setting. This approach is also able to estimate the adaptation effort influencing the component ranking.

Another work on facets was performed by Liao [Liao et al., 1999]. He describes an hybrid similarity scheme using facets, but where each facet can have multiple values. Unlike Prieto's work, this work does not use a conceptual graph to compute the conceptual closeness between facet values. It uses simple list matching. Other difference with Prieto's work and, in this case, similarity with REBUILDER, is the goal of reusing object oriented code and design elements.

RSL [Burton et al., 1987] is a software design system that combines a library of reusable components with several reuse tools. These tools are: a user query subsystem based on natural language and attribute search; a score subsystem that is used to evaluate the retrieved components; and a software computer aided design subsystem used for top-level design specification. Component similarity assessment is an interactive and iterative process between RSL and the user, which is clearly different from REBUILDER where the computation of similarity is automated. Reusing and revising retrieved components is left to the user, while in our approach this can be done by the system or by the designer. It allows reuse of code and design knowledge, and provides several software design tools to work with the retrieved objects. RSL uses automatic indexing of components by scanning design documents and code files for specific labelled reuse component statements. It also provides a functional classification for components using a taxonomy of components.

Borgo [Borgo et al., 1997] developed OntoSeek, which is a software reuse system that uses WordNet for retrieval of object oriented components. OntoSeek uses a graph structure to represent both the query and the components in memory, this representation is called lexical semantic graphs. The retrieval mechanism uses a graph matching algorithm returning the identifiers of all components whose description is subsumed by the query. WordNet is also used for node matching. OntoSeek is essentially for code reuse and does not perform adaptation or verification. Classification and storage of new components is the responsibility of a system analyst. This shows some

similarities with the functions performed by the REBUILDER's KB administrator.

Helm [Helm and Maarek, 1991] describes a system for retrieval of Object-Oriented components based on the class source code, and the class documentation, using natural language queries. In his work, indexing of classes and class documentation is performed automatically by the system. As many software reuse systems described, it only addresses the coding phase of software development.

ROSA [Girardi and Ibrahim, 1994] is a software reuse system that retrieves software components, and works at the code level. This system converts natural language queries, given by the designer, into frames, which are the structures used internally by the system. It's classification mechanism uses a grammar to parse software descriptions, transforming them into frames. Retrieval is based on the similarities of the semantic frames, and on the matching of the noun phrases in the semantic structures using natural language. It is also able to extract indexes from software components, which are then used to index these components in the software repository. ROSA uses a knowledge base that comprises: a classification scheme; WordNet for several tasks, including natural language processing and component classification; and the software repository, which comprises the software components described by frames. The system does not perform adaptation or verification.

Burg and Riet [Burg and Riet, 1997] defended that intelligent CASE systems can benefit from linguistics. They describe a CASE system (COLOR-X) that extracts the structure of the domain model from textual documents, which describe the domain requirements. This analysis consists on parsing the texts and retrieving the word meanings corresponding to the concepts in the model. COLOR-X works with a specific modelling language developed by the same team. Retrieval is based on natural language queries, and adaptation is a composition process guided by model simulation. This simulation is performed using Prolog rules created to simulate the software models that were developed. Starting with a natural language text, COLOR-X is able to generate the code that corresponds to the design models, but with the supervision of the designer that has a crucial role during the process. It works at the design and coding levels of software development. COLOR-X uses a knowledge base comprising a software component repository, WordNet as a lexicon and Prolog rules used for simulation. One of the disadvantages of this system in relation to REBUILDER, is

that, it does not use a generally accepted modelling language like UML.

LaSSIE [Devanbu et al., 1991] is a knowledge-based system that helps programmers searching for useful information in large software systems, attenuating the problem of complexity and invisibility[3] in the reuse of large system components. LaSSIE's application domain is telecommunications software. It uses a knowledge base and a semantic analysis algorithm able to retrieve software components. It provides an interactive interface that facilitates the access to the software through several semantically-based views. It also provides a natural language query interface to the designer, converting the designer query into frames. Knowledge is represented by frames, and the knowledge base comprises a software repository and an indexing structure for reusable components. LaSSIE works at the coding level and is tailored to object oriented languages.

Bassett's [Bassett, 1987, Bassett, 1997] work on software reuse is based on frames. As in other systems described here, software components are described and indexed by frames. His system is intended for procedural code and it addresses the coding phase.

Table 6.6 shows a comparison of the main characteristics of the software reuse systems presented in this section. Notice that all the systems focus on code reuse, except REBUILDER. Some of these systems also perform design reuse, but this is a secondary goal, which is a sub product of code reuse, or the systems deal with design components in the same way they deal with code. Most of the systems cope with object-oriented programming, with the exception of the first two systems (RSL and Prieto's system), which have the goal of reusing any kind of software component. Most of these systems only perform retrieval, which enables them to use a representation formalism easy to manipulate, like attribute/value pairs or frames. More complex representations like in Borgo's approach, which uses graphs to represent code, are not fully explored in our opinion. Another complex representation is the one used in COLOR-X, which enables adaptation using a code generator. In our case, the class diagram representation enables to work with several reasoning mechanisms as we have shown in this thesis.

Regarding retrieval, there are four main approaches for retrieval: attribute-based;

[3]The term invisibility is used in LaSSIE's work to describe the difficulty of the software developers to be aware of all the modules and parts of a large software system.

semantic-based; natural language-based; and structural-based. Attribute-based retrieval, which uses attributes to search the code repository. This retrieval method does not discriminate between functional, structural or behavior attributes. That can be an advantage due to the unrestricted nature of retrieval. But it can also be a limitation for the reasoning mechanisms that deal with the attributes, because they do not know each attribute's type. The second retrieval type is semantic retrieval, which uses functional classification of components to select relevant candidates. One limitation is that there must be a strong domain theory being semantic retrieval, especially in component classification and indexing. The next retrieval type is based on natural language, which most of the time uses keywords and senses to index cases. This retrieval approach can become cumbersome due to the tasks involved in natural language processing. Finally there is structural retrieval, which supports retrieval on component structure, and is more complex than other retrieval types. Due to the highly structured case representation of REBUILDER, it uses functional retrieval as the first criteria for selecting relevant candidates, and then uses functional, structural and behavioral similarity to rank candidates.

6.5 Systems that deal with Analogical Retrieval

This section describes some of the systems that deal with retrieval for analogical reasoning. Since it is an important aspect in REBUILDER and is the focus of one experimental study, we decided to add this section so that the reader can have an idea about other systems addressing the same subject.

Gentner and Forbus [Gentner and Forbus, 1991] have developed the MAC/FAC (Many Are Called/Few Are Chosen) approach to analogy retrieval. It comprises a two-stage structural retrieval model, where in the MAC phase a crude idea of the case structural content is used to retrieve a broad range of cases from memory. Then, in the FAC phase, a detailed structural similarity metric is used to eliminate several cases, and to rank the ones that are chosen.

The Holographic Reduced Representations (HRR) approach [Plate, 1994] combines semantic similarity and structural similarity. The semantic similarity is based

Table 6.6: Comparison table for the main characteristics of software reuse systems.

Name	Application Domain	Knowledge Representation	Candidate Selection	Adaptation
RSL	Component Reuse [Code+Design, All]	Attribute/value pairs	Natural language and attribute search	None
Prieto-Diaz	Code Reuse [Code, All]	Attribute/value pairs	Attribute similarity based on semantic distance	None
Liao	Software Reuse [Code+ Design, Object-Oriented]	Attribute/value pairs	Attribute similarity	None
Borgo	Component Reuse [Code, Object-Oriented]	Graph-based representation	Graph matching retrieval algorithm	None
Helm	Code Reuse [Code, Object-Oriented]	Code representation	Natural language retrieval	None
Girardi (ROSA)	Code Reuse [Code, Object-Oriented]	Frame-based representation	Semantic and structural retrieval	None
Burg (COLOR-X)	ICASE tool [Code+ Design, Object-Oriented]	Specific modelling language	Natural language retrieval	Code generator
Lassie	Telecommunications domain [Code, Object-Oriented]	Frame-based representation	Semantic-based retrieval	None
Bassett	Code Reuse [Code, Procedural]	Frame-based representation	Semantic and structural retrieval	None
REBUILDER	UML Diagram Reuse [Design, Object-Oriented]	Graph-based (class diagrams)	Function, behavior and structure-based	Analogy, CBR and Design Patterns

on the common semantic primitives of objects, while the structural similarity is accessed using the various roles of each object. This structural similarity is not a complete assessment of the structural similarity, but a superficial one.

An approach using constraint satisfaction (ARCS - Analogical Retrieval by Constraint Satisfaction [Thagard et al., 1990]) applies three similarity mechanisms through the creation of a constraint network. These mechanisms are: lexical similarity, structural similarity, and pragmatic similarity.

RADAR (Retrieving Analogies utilizing Derived AttRibutes) developed by Crean and O'Donoghue [Crean and O'Donoghue, 2001], was created to retrieve semantically distant analogies in a computationally tractable manner. It is based only on structure mapping (systematicity), and the main concern of this approach was that distant analogies would not be rejected by the semantic similarity. We have adapted some of the ideas in this work to REBUILDER, and tested them.

REBUILDER takes inspiration from these works, and especially from RADAR. We have implemented a candidate selection metric for selection of analog cases, based on RADAR work, which states that structure is more important for analogy than semantic similarity. Section 5.3.3 describes the experimental work performed with the idea of RADAR in mind, adapted to REBUILDER's situation. The MAC/FAC work is very different from our work because it only uses structural similarity to select candidate sources for analogy. ARCS uses other type of knowledge for retrieval, like lexical or pragmatic, which are not used in REBUILDER. HRR is the most similar work to REBUILDER, because it combines semantic and structural similarity. REBUILDER also adds up behavior similarity.

6.6 Design Patterns

There are some research works in the field of software design patterns that have common aspects with our work. This section describes these works and compares them with our approach. It is important to say that research on software design patterns has three main lines of action: (1) discovery of new patterns, (2) classification, description and representation of patterns, and (3) application of patterns. In our work, we deal primarily with the application of patterns, but we also address the

issue of pattern representation. The systems described here deal with the application of patterns.

Eden et. al. [Eden et al., 1997] proposed an approach to the specification of design patterns, and a prototype tool that extensively automates their application. This approach considers design patterns as programs that manipulate other programs, thus they are viewed as metaprograms. Eden's approach does not automate the entire process of pattern application, since it is the designer that has to select which pattern to apply. In our approach this step is automated. Another difference is that Eden applies patterns at the code level, while we apply them at the design level.

Tokuda and Batory [Tokuda and Batory, 1995] also present an approach in which patterns are expressed in the form of a series of parameterized program transformations applied to software code (refactorings). Refactorings are basic program transformations, which are at a lower level than REBUILDER's pattern operations. In our case a pattern has only an operator, but an operator can use another operator. In Tokuda's work several refactorings can express one pattern. Like Eden's work, this work does not address the automation of pattern selection. It is important to stress that the application of refactorings has to be assisted by the user.

Other works on specifying design patterns and automating its application are presented by Bär [Bär et al., 1999] and Cinnéide [Cinnéide and Nixon, 1999]. These works also automate the application of design patterns, but do not select which pattern to apply. This must be performed by the designer. Both works deal with design modification instead of code modification.

Cinnéide's work starts with the user selection of a design pattern. After this, the precursor for the selected pattern must be chosen. A precursor is defined as a design structure that expresses the intent of a design pattern. This concept is very similar to our concept of pattern participant, but is more specific. After the precursor is chosen, the selected pattern is decomposed into a sequence of minipatterns, which are a design motif that occurs frequently (similar to a design pattern but at a lower level). Then this sequence is applied in the precursor 'point' resulting into the design transformation.

6.7 Word Sense Disambiguation

Word sense disambiguation (WSD) is the process of identifying the correct meaning of a word. In REBUILDER this is an important task, because this is the process that associates a synset to a software object. This synset is then used for several purposes, from classification to similarity computation. WSD involves two main steps (for each word): establishing the word possible senses; devise a method of assigning the correct sense to the word. Most of the recent work on WSD uses a set of pre-defined senses from dictionaries or already created from a thesaurus, as a solution for step one. The systems that deal with the second step, can be classified into three main categories: knowledge-driven, data-driven (also called corpus-based), or hybrid (combining both approaches).

A work following a knowledge-driven approach was presented by Nastase and Szpakowicz [Nastase and Szpakowicz, 2001] involving disambiguation of word senses in Roget's Thesaurus. They use the information in WordNet to disambiguate word senses, based on word lists of neighbor senses. Then these word lists are compared yielding a disambiguation score that can be used to determine which sense to use. We have implemented this method in REBUILDER.

Wilks and Stevenson [Wilks and Stevenson, 1996] propose another knowledge-driven method based on part-of-speech tagging[4]. Their main claim is that part-of-speech tagging has some connection to semantic disambiguation. This method was not implemented in REBUILDER, because REBUILDER does not include a part-of-speech tagger.

Most of the recent methods combine both approaches - knowledge and data driven. An hybrid method was presented by Yarowsky [Yarowsky, 1992], he supports the WSD method on statistical models for Roget's Thesaurus categories. These categories can be regarded as conceptual classes which tend to correspond to senses. A version of this method is implemented in REBUILDER under the name of semantic vectors. Instead of using Roget's Thesaurus categories, it uses WordNet top noun senses.

Kwong [Kwong, 2001] has recently presented a WSD study using WordNet and Roget's Thesaurus testing four different WSD methods, which are then combined

[4]Part-of-speech tagging is the process of associating a lexical category to the words in a text.

with the entry sense ordering method to provide the best results. This combination enhances significantly the accuracy of other methods, showing that the combination of the two approaches (knowledge and data-driven) can be very positive. REBUILDER implements a method that combines several WSD methods, called method combination, but surprisingly, experimental results are not very promising, either to our domain (diagram disambiguation) or to more general tests like SemCor (see section 5.8) .

Li et. al. [Li et al., 1995] propose an algorithm for WSD based on WordNet, which uses relations between verb and noun. Their method is applied to nouns, that are objects of verbs in sentences. The method is based on semantic similarity between concepts and is computed using WordNet. The difference concerning our system, is that REBUILDER only uses nouns, since it works only with entity names. A possibility to use verbs within our approach, to aid the disambiguation of software object names, would be to use relation's names in UML. Unfortunately, from our experience, designers often do not give names to relations.

Smeaton and Quigley [Smeaton and Quigley, 1996] describe an information retrieval system for image retrieval based on image captions. They use two disambiguation methods, both based on a semantic distance between words. This semantic distance uses WordNet, and the probability of the Most Specific Common Abstraction (MSCA) between synsets, occurring in a large text corpus.

Michalcea and Moldovan [Mihalcea and Moldovan, 1998] propose a WSD method based on the idea of semantic density between words. They use WordNet as a general ontology and Internet as a source for raw corpora providing statistical information for word associations. Their approach is targeted for verb-noun pairs and they define semantic density between words as: the number of common words that are within a semantic distance of two or more words. The closer the semantic relationship between two words, the higher the semantic density between them. They also present an iterative WSD method [Mihalcea and Moldovan, 1999], based on WordNet and SemCor [Miller et al., 1994]. This method disambiguates words iteratively based on several procedures that use word relations in WordNet.

Rigau et. al. [Rigau et al., 1997] present a WSD method based on the combination of several unsupervised algorithms. Their approach is driven by the assumption

that resolution of lexical ambiguity can be successful combining several information sources and techniques. In REBUILDER, we use a method which combines several WSD methods, but experimental results (see subsection 5.8.2) were not encouraging. The experimental results presented in Rigau's work show an improvement in disambiguation results, with the integration of several WSD methods. Though in some results, the individual methods are better than the score for the combination approach.

Resnik [Resnik, 1995a, Resnik, 1995b] uses WordNet and data on word frequency as a basis for a WSD method that takes into account the semantic similarity of synsets. A basic assumption of this method, is that the similarity of two concepts is the extent to which they share common information. This is one of the methods implemented in REBUILDER.

Gale [Gale et al., 1992] provides an estimate for upper and lower bounds of WSD method performance, and stresses that method comparison is very difficult, since experimental set-ups are very different. He identifies the lower bound of 68% and an upper bound of 96.8%. Obviously, these limits depend not only on the particular experimental design, but also on the human judgment made on the classification. This is clearly seen in the experiments performed using several WSD methods implemented in REBUILDER (see subsection 5.8.2). With methods applied in similar circumstances, the accuracy reported by the authors of these methods is much lower.

6.8 Discussion

From the several issues to be addressed by an Intelligent CASE (ICASE) tool, we focus on those related with knowledge use and reasoning needs. In this chapter we follow this framework to make a comparative study on different systems versus REBUILDER.

Most of the functionalities present in ICASE tools complement or assist a human in designing a system. This is performed with the system carrying out the routine tasks, relieving the human designer to more creative tasks. But there are also functionalities that can only be accomplish in an automated fashion, especially those involving huge amount of knowledge or data. For instance, searching a KB with

hundreds or thousand of objects is an impossible (or at least impractical) task for humans.

The basic functionality that an ICASE tool should have, is the ability to suggest alternative designs. In CBR and analogy-based systems this can be accomplished by retrieval of similar designs and leaving to the designer the adaptation phase, or automating this step. In REBUILDER the designer has the flexibility to choose, what s/he prefers. Most of the design systems presented in this chapter perform adaptation. This functionality is important so that the designer can explore the design space, evaluating the design alternatives.

Another important issue related to design adaptation, is that most of the times, reuse implies adaptation, since the common scenario is that the most similar design needs modification to be reused. This makes adaptation a very important task, but it is also a knowledge intensive process requiring a lot of expertise to be successfully performed. From the experimental results presented in chapter 5, it can be seen that adaptation is not fail-proof and that it still needs human intervention. A possible solution in helping the designer reviewing the generated design is to complement the adaptation strategies with a verification module, as it is performed in REBUILDER.

Another basic functionality of an ICASE tool is to structure the design experiences and to make them accessible for future use. This is accomplished by structured design repositories in a way that they enable classification and storage of software designs, and also allow efficient retrieval of these designs. In REBUILDER and in most of the other systems, this is accomplished through a case library that stores designs as cases, ready for later use. REBUILDER goes further, because it organizes these cases using general knowledge in the form of an ontology. This enables inter-domain retrieval, which is not performed by other CBR systems described here. The design repository is also important for the company, because it represents a corporative memory that can be reused. This prevents the loss of this know-how when an experienced designer leaves the company.

Designers are the main focus for an ICASE tool, but they can also be seen as knowledge sources. In REBUILDER, the designer provides several types of knowledge: design cases, verification cases, and software design pattern application cases.

These cases can be acquired from the system's reasoning, but they can also be acquired from the interaction with the designer. All the other systems learn only design cases. Taking advantage of the designer interaction to learn new knowledge is an important functionality for an ICASE tool, which enables the improvement of the system's reasoning capabilities.

An ICASE tool must also support evolutional and incremental development of the system's design, as well as maintenance. Flexibility is a key issue here. The system must be flexible enough to provide the designer with several tools. Each tool can be used at the appropriate stage, allowing an incremental evolution of the design. REBUILDER integrates such framework by providing the designer with several cognitive functions, selectable at any point in the design. Some of these functions are: retrieval, adaptation, verification, and evaluation. For design maintenance, our system provides the application of software design patterns to evolve and restructure the system's design. Other systems do not provide alternatives in this domain.

Another important aspect is the capability to represent, understand and visualize artifact semantics. These are important aspects for the designer, helping her/him understanding the reusable components, making possible the communication with the system. Some representation mechanisms are unsuitable for designers, two main reasons for this are: they are not schematic enough, or they are not familiar to the designer. In our approach, this issue is solved using UML. Being one of the most used software modelling languages, and integrating several visual diagrams capable of describing a system along many views, it is a suitable representation formalism for the designer and for the system. We also integrate a natural language component through WordNet. Most of the other systems use specific modelling languages, which makes them difficult to use, due to the human-system communication barrier.

Other functionalities that an ICASE should integrate are: enable reverse engineering, code generation, and integration with the programming environment. These are more programming-centered issues, because they are not necessarily needed for design reuse, though they are important for the system's usage. These extra functionalities can be implemented in a more intelligent way, automating several boring and time consuming tasks that are performed by designers. Human programmers tend to be reluctant to modify code that they didn't write, and this can be a way to make them

more comfortable about code revision or modification. The same applies to design. This is one future research direction for REBUILDER.

Chapter 7

Conclusions and Future Work

This chapter presents the main contributions of this thesis, which are the result of a three-year research project. The main areas of contribution are: Case-Based Reasoning, Software Reuse, Intelligent CASE tools, Word Sense Disambiguation, Analogy and Creative Design Systems. Along the development of REBUILDER several new research directions were also identified, but due to the dimension of the research necessary, they have been postponed for future work. We present new research challenges and improvements that can be performed on REBUILDER.

7.1 Thesis Contributions

REBUILDER has been a test ground for many ideas and a challenge that has touched several areas. Our initial goal was to develop a software system that could reuse previous designs in new projects. This idea matured and evolved into a system with a client-server architecture, based on CBR for reasoning and knowledge management. It is also viewed as a tool that enables reuse at corporate level, working as a knowledge repository. Communication aspects between designer and system also received attention. This was accomplished through the use of UML and a first step into incorporating natural language understanding, in the form of word sense disambiguation for software object names.

The main contributions of our work are:

- A client/server architecture supported on a CBR framework, as well as the way the different modules interact are one of the main contributions of our work.

This provides an approach for integration of a ICASE tool with a knowledge management tool for software design.

- An UML representation. Instead of choosing a computer-centered case representation formalism we have used a user-centered approach, which lead us to choose UML as the representation formalism. We show that it is possible to use UML for reasoning purposes and we have presented a way of integrating it into a CBR system.

- An indexing scheme based on WordNet. Regarding case indexing and retrieval we have made two contributions: the indexing scheme based on WordNet that indexes not only the cases but also it's subparts, and the associated retrieval and similarity metrics.

- In the case adaptation problem we have addressed this issue in a broad perspective, trying to explore several alternative approaches:

  - In the analogical reasoning for case adaptation, our main contributions are the way WordNet is used for choosing analogue source candidates, the mapping algorithms and the experimental work about creative design.

  - Another adaptation mechanism is the design composition, which can generate new solutions from several cases. Our main contribution here is the composition strategies that were developed.

  - The last adaptation strategy is the automatic application of software design patterns. We think that this is one of the strongest contributions of this work, because it is the first approach that fully automates the application of software design patterns, going from the pattern selection to its application.

- A mechanism for solution verification. In the verification of solutions we have contributed in three ways: the definition and use of verification cases, the personalization of the verification knowledge in the form of cases, and the use of WordNet for verification purposes.

- Maintenance policies for the case base. We have integrated several case base maintenance policies, which provide guidance for the KB Administrator in the

selection of the case base contents. Our contribution is the adaptation of these policies to a graph-based representation of cases.

- A word sense disambiguation mechanism. In the problem of word sense disambiguation we have contributed with two new disambiguation methods and experimental work comparing several methods from the WSD literature.

7.2 Future Work

During these three years of work, we have came across several research issues, some of which we addressed because we thought they were within the scope of this thesis, others were left for future work or for other researchers to pursue. We now describe them, organized along research areas.

- The integration of other development phases, in addition to design, into the REBUILDER scope is one of our goals for future work. The analysis and implementation phases are the next steps in this path. The design phase can also be improved by addressing other types of UML diagrams. The integration of code reuse in REBUILDER allows changes made in the code to be represented in the corresponding design model, and vice-versa.

- Another future work on REBUILDER will be the integration of a natural language module, capable of understanding a text in English and yield a class diagram modelling the text contents.

- As a CASE tool, REBUILDER has a reactive behavior. A possible future direction can be the modification of REBUILDER's behavior to a more proactive one. This would help the human-machine communication, because it would model the designer's cognitive states and would try to suggest actions based on it.

- One important improvement would be the integration of specialized knowledge in WordNet, this could be done allowing the KB administrator to insert, delete or modify synsets or relations between synsets. Concerning this issue, we have

already integrated the Java class hierarchy into WordNet, which enables REBUILDER to deal with specific object-oriented classes like: button, file and others.

- New reuse mechanisms can be easily integrated in REBUILDER. Several new ways of adaptation can be used, such as, model-based reasoning, constraint-based reasoning, genetic algorithms and others.

- We would also like to improve the analogy mechanism in order to provide better evaluation guidelines in respect to creative properties of generated designs.

- Verification of classes and interfaces is superficially addressed within our work. The only action performed in order to check class or interface validity is to verify the object name. This could be taken further away, if the verification mechanism could inspect and reason with the class's attributes and methods. The retrieval process used to select verification cases follows a very restrictive view of case usage. This can be explored in future work, especially the development of a similarity metric for verification cases that is able to relax the retrieval constraints.

- Learning could be introduced in case verification. This could be achieved by learning general verification principles from verification cases, using an abstraction process over verification cases and WordNet. Remember that each designer has its own verification case base.

- Further experimental work with the case base maintenance strategies is necessary in order to better characterize their behaviors. The main problem here is to get a case base large enough, which would be a good representative of the domain space. In conjunction with these experiments, we intended to perform a study of the effect of different subsumption criteria and combination of several policies.

- Reasoning quality is highly dependent on a correct object classification, which makes the WSD a crucial part of our system. Although we achieved good results, they are not yet 100% correct, which probably will never be achieved. Notwithstanding, there is a progress margin concerning this issue that justifies

a research investment on this problem, and specially about "diagram disambiguation". The disambiguation of relations' names can also be addressed, with the benefit of the system being able to provide synsets to UML class diagram relations. This could provide more knowledge for reasoning mechanisms to work with. Testing other lexical resources or text corpus with the WSD methods can also be a possibility of improvement of WSD.

- A main future work issue is trying to field test REBUILDER in a software company, which implies the improvement of REBUILDER's prototype. The Goal-Query-Metric methodology [Nick et al., 1999] can be used in the field testing. This could be the starting point for REBUILDER II.

Bibliography

[Aamodt and Plaza, 1994] Aamodt, A. and Plaza, E. (1994). Case–Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1):39–59.

[Althoff, 2001] Althoff, K.-D. (2001). Case-Based Reasoning. In Chang, S. K., editor, *Handbook on Software Engineering and Knowledge Engineering*, volume 1, pag. 549–588. World Scientific.

[Althoff et al., 1997] Althoff, K.-D., Birk, A., Wangenheim, C. G. v., and Tautz, C. (1997). Case–Based Reasoning for Experimental Software Engineering. Technical Report 063.97/E, Fraunhofer IESE.

[Atkinson et al., 2002] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wurst, J., and Zettel, J. (2002). *Component–Based Product Line Engineering with UML*. Addison–Wesley, London.

[Bär et al., 1999] Bär, H., Bauer, M., Ciupke, O., Demeyer, S., Ducasse, S., Lanza, M., Marinescu, R., Nebbe, R., Nierstrasz, O., Richner, T., Rieger, M., Riva, C., Sassen, A.-M., Schulz, B., Steyaert, P., Tichelaar, S., and Weisbrod, J. (1999). The FAMOOS object–oriented reengineering handbook. Technical report, Forschungszentrum Informatik, Software Composition Group, University of Berne.

[Bassett, 1997] Bassett, P. (1997). *Framing Software Reuse: Lessons From the Real World*. Yourdon Press.

[Bassett, 1987] Bassett, P. G. (1987). Frame–Based Software Engineering. *IEEE Software*, 4(July):9–16.

[Bento, 1996] Bento, C. (1996). *Raciocínio Baseado em Casos Descritos e Explicados de Modo Imperfeito*. Ph. d., University of Coimbra, Department of Informatic Engineering.

[Bento et al., 1995] Bento, C., Macedo, L., and Costa, E. (1995). Reasoning with Cases Imperfectly Described and Explained. In Haton, J.-P., Keane, M., and

Manago, M., editors, *Proceedings of the 2nd European Workshop on Advances in Case-Based Reasoning*, volume 984 of *LNAI*, pag. 45–59, Berlin. Springer Verlag.

[Bergmann and Stahl, 1998] Bergmann, R. and Stahl, A. (1998). Similarity Measures for Object-Oriented Case Representations. In Cunningham, P. and Smyth, B., editors, *4th European Conference on Case-Based Reasoning*, LNAI, Dublin, Ireland. Springer.

[Bhatta and Goel, 1992] Bhatta, S. and Goel, A. (1992). Use of Mental Models for Constraining Index Learning in Experience-Based Design. In *Proceedings of AAAI-92 workshop on Constraining Learning with Prior Knowledge*, San Jose, CA, USA.

[Bhatta and Goel, 1993a] Bhatta, S. and Goel, A. (1993a). Learning Generic Mechanisms from Experiences for Analogical Reasoning. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, Boulder, CO, USA.

[Bhatta and Goel, 1993b] Bhatta, S. and Goel, A. (1993b). Model-Based Learning of Structural Indices to Design Cases. In *Proceedings of IJCAI-93 workshop on Reuse of Designs: An Interdisciplinary Cognitive Approach*, France.

[Bhatta and Goel, 1997a] Bhatta, S. and Goel, A. (1997a). An Analogical Theory of Creativity in Design. In *International Conference on Case-Based Reasoning (ICCBR 97)*, Providence - Rhode Island, USA. Springer-Verlag.

[Bhatta and Goel, 1997b] Bhatta, S. and Goel, A. (1997b). Design Patterns; A Computational Theory of Analogical Design. In *International Joint Conference on Artificial Intelligence (IJCAI'97)*.

[Boden, 1990] Boden, M. (1990). *The Creative Mind: Myths and Mechanisms.* Weidenfeld and Nicolson, London.

[Boehm, 1988] Boehm, B. (1988). *A Spiral Model of Software Development and Enhancement*. IEEE Press.

[Borgo et al., 1997] Borgo, S., Guarino, N., Masolo, C., and Vetere, G. (1997). Using a Large Linguistic Ontology for Internet-Based Retrieval of Object-Oriented Components. In *9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, pag. 528–534, Madrid, Spain. Knowledge Systems Institute, Illinois.

[Brown, 1989] Brown, R. (1989). Creativity: What are We to Measure? In Glover, J., Ronning, R., and Reynolds, C., editors, *Handbook of Creativity*. Plenum Press.

[Burg and Riet, 1997] Burg, J. F. M. and Riet, R. P. v. d. (1997). Trully Intelligent CASE Environments profit from Linguistics. In *9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, pag. 407–414, Madrid, Spain. Knowledge Systems Institute, Illinois.

[Burton et al., 1987] Burton, B. A., Aragon, R. W., Bailey, S. A., Koehler, K. D., and Mayes, L. A. (1987). The Reusable Software Library. *IEEE Software*, 4(July 1987):25–32.

[Bylander and Chandrasekaran, 1985] Bylander, T. and Chandrasekaran, B. (1985). Understanding Behavior Using Consolidation. In Joshi, A., editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pag. 450–454, Los Angeles, CA. Morgan Kaufmann.

[Cinnéide and Nixon, 1999] Cinnéide, M. and Nixon, P. (1999). A Methodology for the Automated Introduction of Design Patterns. In *IEEE International Conference on Software Maintenance*, Oxford, England. IEEE.

[Colton et al., 2001] Colton, S., Pease, A., and Ritchie, G. (2001). The Effect of Input Knowledge on Creativity. In *International Conference on Case-Based Reasoning (ICCBR'01) Workshop on Creative Systems: Approaches to Creativity in Artificial Intelligence and Cognitive Science*, Vancouver, Canada. Technical Report of the Navy Center for Applied Research in Artificial Intelligence (NCARAI) of the Naval Research Laboratory (NRL).

[Coulange, 1997] Coulange, B. (1997). *Software Reuse*. Springer Verlag, London.

[Coulon and Gebhardt, 1994] Coulon, C.-H. and Gebhardt, F. (1994). Evaluation of Retrieval Methods in Case-Based Design. In Keane, M., Haton, J.-P., and Manago, M., editors, *2nd Workshop on Case-Based Reasoning, EWCBR'94*, pag. 283–291, Chantilly, Paris, France. Acknosoft Press.

[Crean and O'Donoghue, 2002] Crean, B. and O'Donoghue, D. (2002). RADAR: Finding Analogies using Attributes of Structure. In *Proceedings of the 13th Irish Conference on Artificial Intelligence and Cognitive Science (AICS'02)*, pag. 20–27, Limerick, Ireland. Springer-Verlag.

[Crean and O'Donoghue, 2001] Crean, B. P. and O'Donoghue, D. (2001). Features of Structural Retrieval. In *IASTED - International Symposia, Applied Informatics*, pag. 295–300, Innsbruck, Austria.

[Dasgupta, 1994] Dasgupta, S. (1994). Creativity, Invention and the Computational Metaphor: Prolegomenon to a Case Study. In Dartnall, T., editor, *Artificial Intelligence and Creativity*. Kluwer Academic Publishers.

[Dave et al., 1991] Dave, B., Schmitt, G., Shih, S., Bendel, L., Faltings, B., Smith, I., Hua, K., Bailey, S., Ducret, J., and Jent, K. (1991). Case-Based Spatial Design Reasoning. In *First European Workshop on Case-Based Reasoning*.

[Devanbu et al., 1991] Devanbu, P., Branchman, R. J., Selfridge, P. G., and Ballard, B. W. (1991). LaSSIE: A Knowledge-based Software Information System. *Communications of ACM*, 34(5):34–49.

[Domeshek and Kolodner, 1993] Domeshek, E. A. and Kolodner, J. L. (1993). Finding the points of large cases. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, 7(2):87–96.

[Dudani, 1976] Dudani, S. (1976). The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man and Cybernetics*, 6(4):325–327.

[Duffy and MacCallum, 1989] Duffy, A. H. and MacCallum, K. J. (1989). Computer Representation of Numerical Expertise for Preliminary Ship Design. *Marine Technology*, 2(26):289–302.

[Eden et al., 1997] Eden, A., Gil, J., and Yehudai, A. (1997). Automating the Application of Design Patterns. *Journal of Object Oriented Programming*, 1(May).

[Fernández-Chamizo et al., 1996] Fernández-Chamizo, C., González-Calero, P., Gómez-Albarrán, M., and Hernández-Yánez, L. (1996). Supporting Object Reuse through Case-Based Reasoning. In Smith, I. and Faltings, B., editors, *Third European Workshop on Case-Based Reasoning (EWCBR'96)*, volume 1168, pag. 150–163, Lausanne, Suisse. Springer-Verlag.

[Fouqué and Matwin, 1993] Fouqué, G. and Matwin, S. (1993). Compositional Software reuse with Case-Based Reasoning. In *9th Conference on Artificial Intelligence for Applications (CAIA'93)*, Orlando, FL, USA. IEEE Computer Society Press.

[Francis and Kuçera, 1982] Francis, W. N. and Kuçera, H. (1982). *Frequency Analysis of English Usage: Lexicon and Grammar*. Houghton Mifflin, Boston.

[Gale et al., 1992] Gale, W., Church, K., and Yarowsky, D. (1992). Estimating Upper and Lower Bounds on the Performance of Word-Sense Disambiguation Programs. In *30th Annual Meeting of the ACL*, pag. 249–256. ACL.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading.

[Gebhardt et al., 1997] Gebhardt, F., Voss, A., Gräther, W., and Schmidt-Belz, B. (1997). *Reasoning with Complex Cases*. Number 393 in The Kluwer Series in Engineering and Computer Science. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands.

[Gentner, 1983] Gentner, D. (1983). Structure Mapping: A Theoretical Framework for Analogy. *Cognitive Science*, 7(2):155–170.

[Gentner and Forbus, 1991] Gentner, D. and Forbus, D. (1991). MAC/FAC : A model of Similarity-based Retrieval. In *13th Conference of the Cognitive Science Society*, pag. 504–509.

[Gero, 1994a] Gero, J. (1994a). Computational Models of Creative Design Processes. In Dartnall, T., editor, *Artificial Intelligence and Creativity*. Kluwer Academic Publishers.

[Gero, 1994b] Gero, J. (1994b). Introduction: Creativity and Design. In Dartnall, T., editor, *Artificial Intelligence and Creativity*. Kluwer Academic Publishers.

[Gero and Maher, 1993] Gero, J. and Maher, M. L. (1993). *Modelling Creativity and Knowledge-Based Creative Design*. Lawrence Erlbaum Associates, Sydney.

[Girardi and Ibrahim, 1994] Girardi, M. R. and Ibrahim, B. (1994). A Similarity Measure for Retrieving Software Artifacts. In *6th International Conference on Software Engineering and Knowledge Engineering*, pag. 478–485, Jurmala, Latvia.

[Goel, 1991] Goel, A. (1991). A Model-Based Approach to Case Adaptation. In *13th Annual Conference on Cognitive Science Society (CogSci'91)*, Chicago, Illinois, USA.

[Goel, 1992] Goel, A. (1992). Representation of Design Functions in Experience-Based Design. In Brown, D., Waldron, M., and Yosnikawa, H., editors, *Intelligent Computer Aided Design*. Elsevier Science.

[Goel et al., 1997a] Goel, A., Bhatta, S., and Stroulia, E. (1997a). Kritik: An Early Case-Based Design System. In Maher, M. L. and Pu, P., editors, *Issues and Applications of Case-Based Reasoning to Design*. Lawrence Erlbaum Associates.

[Goel et al., 1997b] Goel, A., Bhatta, S., and Stroulia, E. (1997b). KRITIK: An Early Case-Based Design System. In Maher, M. L. and Pu, P., editors, *Issues and Applications of Case-Based Reasoning in Design*, pag. 87–132, Mahwah, NJ. Lawrence Erlbaum Associates.

[Goel et al., 1993] Goel, A., Malkawi, A., Pearce, M., and Liu, K. (1993). A Cross-Domain Experiment in Case-Based Design Support: ArchieTutor. In Leake, D., editor, *Papers from the AAAI-93 Workshop on Case-Based Reasoning*, pag. 111–117, Washington, D.C. AAAI Press.

[Gomes and Bento, 1997a] Gomes, P. and Bento, C. (1997a). A Case-Based Approach for Elaboration of Design Requirements. In *International Conference on Case-Based Reasoning (ICCBR 97)*, Providence - Rhode Island, USA. Springer-Verlag.

[Gomes and Bento, 1997b] Gomes, P. and Bento, C. (1997b). A Retrieval Method for Exploration of a Case Memory. In *Portuguese Conference on Artificial Intelligence*, Coimbra, Portugal. Springer-Verlag.

[Gomes and Bento, 1998] Gomes, P. and Bento, C. (1998). Experiments on a Memory Structure Supporting Creative Design. In *14th Brazilian Symposium on Artificial Inteligence (SBIA'98)*, Porto Alegre, Brazil. Springer.

[Gomes and Bento, 1999a] Gomes, P. and Bento, C. (1999a). Automatic Conversion of VHDL Programs into Cases. In *International Conference on Case-Based Reasoning (ICCBR'99) Workshop: Practical Case-Based Reasoning Strategies for Building and Maintaining Corporate Memories.*, Seeon Monastery, Munich, Germany.

[Gomes and Bento, 1999b] Gomes, P. and Bento, C. (1999b). Converting Programs into Cases for Software Reuse. In *International Joint Conference on Artificial Intelligence (IJCAI 99) Workshop: Automating the Construction of Case-Based Reasoners*, Stockholm, Sweden.

[Gomes and Bento, 2001] Gomes, P. and Bento, C. (2001). A Case Similarity Metric for Software Reuse and Design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 15(1):21–35.

[Gomes et al., 1998] Gomes, P., Bento, C., and Gago, P. (1998). Learning to verify design solutions from failure knowledge. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(2):107–115.

[Gomes et al., 1996] Gomes, P., Bento, C., Gago, P., and Costa, E. (1996). Towards a Case-Based Model for Creative Processes. In *12th European Conference on Artificial Intelligence (ECAI'96)*, Budapest, Hungary. John Willey & Sons.

[González and Fernández, 1997] González, P. A. and Fernández, C. (1997). A Knowledge-Based Approach to Support Software Reuse in Object-oriented Libraries. In *9th International Conference on Software Engineering and Knowledge*

Engineering, SEKE'97, pag. 520–527, Madrid, Spain. Knowledge Systems Institute, Illinois.

[González-Calero et al., 1999] González-Calero, P. A., Gómez-Albarrán, M., and Díaz-Agudo, B. (1999). A substitution-based adaptation model. In *Challenges for Case-Based Reasoning - Proc. of the ICCBR'99 Workshops*, Germany. Centre for Learning Systems and Applications - Dpt. of Computer Science - University of Kaiserslautern.

[Hall, 1989] Hall, R. (1989). Computational approaches to analogical reasoning: A comparative analysis. *Artificial Intelligence*, 39(1):39–120.

[Harandi and Bhansali, 1998] Harandi, M. and Bhansali, S. (1998). Program Derivation Using Analogy. In Sridharan, N., editor, *Eleventh International Joint Conference on Artificial Intelligence*, pag. 389–394, Detroit, Michigan, USA. Morgan Kaufmann Publishers, San Mateo, California.

[Hart, 1968] Hart, P. E. (1968). The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14:515–516.

[Helm and Maarek, 1991] Helm, R. and Maarek, Y. S. (1991). Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries. In Paepcke, A., editor, *Object-Oriented Programming Systems, Languages, and Applications*, pag. 47–61, Phoenix, AZ USA. ACM Press.

[Holyoak and Thagard, 1989] Holyoak, K. J. and Thagard, P. (1989). Analogical Mapping by Constraint Satisfaction. *Cognitive Science*, 13:295–355.

[Ide and Veronis, 1998] Ide, N. and Veronis, J. (1998). Introduction to the Special Issue on Word Sense Disambiguation: The State of the Art. *Computational Linguistics*, 24(1):1–40.

[Jeng and Cheng, 1993] Jeng, J.-J. and Cheng, B. (1993). Using Analogy and Formal Methods for Software Reuse. In *IEEE 5th International Conference on Tools with AI*.

[Johnson, 1997] Johnson, R. E. (1997). Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42.

[Kolodner, 1993] Kolodner, J. (1993). *Case-Based Reasoning*. Morgan Kaufman.

[Kolodner and Wills, 1993] Kolodner, J. and Wills, L. (1993). Case-Based Creative Design. In *AAAI Spring Symposium on AI+Creativity*, Stanford, CA, USA.

[Krampe and Lusti, 1997] Krampe, D. and Lusti, M. (1997). Case-Based Reasoning for Information System Design. In *International Conference on Case-Based Reasoning (ICCBR 97)*, pag. 63–73, Providence - Rhode Island, USA. Springer-Verlag.

[Kwong, 2001] Kwong, O. Y. (2001). Word Sense Disambiguation with an Integrated Lexical Resource. In *NAACL Workshop on WordNet and Other Lexical Resources*, Pittsburgh, USA.

[Leake and Wilson, 2000] Leake, D. and Wilson, D. (2000). Remembering Why to Remember: Performance-Guided Case Base Maintenance. In *Proceedings of the European Workshop on Case-Based Reasoning (EWCBR-00)*, LNAI, pag. 161–172, Berlin. Springer.

[Li et al., 1995] Li, X., Szpakowicz, S., and Matwin, S. (1995). A WordNet-based Algorithm for Word Sense Disambiguation. In *14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pag. 1368–1374, Montreal, Canada.

[Liao et al., 1999] Liao, S. Y., Cheung, L. S., and Liu, W. Y. (1999). An Object-Oriented System for the Reuse of Software Design Items. *Journal of Object-Oriented Programming*, 11(8, January 1999):22–28.

[Liebowitz, 1999] Liebowitz, J. (1999). *Knowledge Management Handbook*. CRC Press.

[Maher, 1996] Maher, M. L. (1996). Developing Case-Based Reasoning for Structural Design. *IEEE Expert*, 11(3, June 1996).

[Maher et al., 1995] Maher, M. L., Balachandran, M., and Zhang, D. (1995). *Case-Based Reasoning in Design*. Lawrence Erlbaum Associates.

[Maiden and Sutcliffe, 1992] Maiden, N. and Sutcliffe, A. (1992). Exploiting Reusable Specifications Through Analogy. *Communications of the ACM*, 35(4):55–64.

[McKenna and Smyth, 2000] McKenna, E. and Smyth, B. (2000). Competence-Guided Case Base Editing Techniques. In *Proceedings of the European Workshop on Case-Based Reasoning (EWCBR-00)*, LNAI, pag. 186–197, Berlin. Springer.

[Mihalcea and Moldovan, 1999] Mihalcea, R. and Moldovan, D. (1999). A method for Word Sense Disambiguation of unrestricted text. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL-99)*, Maryland, NY, USA.

[Mihalcea and Moldovan, 1998] Mihalcea, R. and Moldovan, D. I. (1998). Word Sense Disambiguation based on Semantic Density. In *COLIN/ACL Workshop on Usage of WordNet in Natural Language Processing Systems*, Montreal, Canada.

[Miller et al., 1990] Miller, G., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. J. (1990). Introduction to WordNet: an on-line lexical database. *International Journal of Lexicography*, 3(4):235 – 244.

[Miller et al., 1994] Miller, G., Chodorow, M., Landes, S., Leacock, C., and Thomas, R. (1994). Using a semantic concordance for sense identification. In *Proceedings of ARPA Human Language Technology Workshop*, pag. 240–243.

[Minton, 1990] Minton, S. (1990). Quantitative Results Concerning the Utility of Explanation-Based Learning. *Artificial Intelligence*, 42(2-3):363–391.

[Nastase and Szpakowicz, 2001] Nastase, V. and Szpakowicz, S. (2001). Word Sense Disambiguation in Roget's Thesaurus Using WordNet. In *NAACL 2001 Workshop on WordNet and other Lexical Resources*, pag. 17–22, Pittsburg, USA.

[Netten and Vingerhoeds, 1996] Netten, B. and Vingerhoeds, R. (1996). Incremental Adaptation for Conceptual Design in EADOCS. In *ECAI96 Workshop on Adaptation in Case–Based Reasoning*, Budapest, Hungary.

[Nick et al., 1999] Nick, M., Althoff, K.-D., and Tautz, C. (1999). Facilitating the Practical Evaluation of Organizational Memories Using the Goal-Question-Metric Technique. In *Twelfth Workshop on Knowledge Acquisition, Modeling and Management, KAW'99*, Banff.

[O'Donoghue and Crean, 2002] O'Donoghue, D. and Crean, B. (2002). Searching for Serendipitous Analogies. In *European Conference on Artificial Intelligence ECAI'02 Workshop: 2nd Workshop on Creative Systems*, Lyon, France.

[Partridge and Rowe, 1994] Partridge, D. and Rowe, J. (1994). *Computers and Creativity*. Intellect Books.

[Pease, 2001] Pease, A. (2001). Evaluating Machine Creativity. In *International Conference on Case-Based Reasoning (ICCBR'01) Workshop on Creative Systems: Approaches to Creativity in Artificial Intelligence and Cognitive Science*, Vancouver, Canada. Technical Report of the Navy Center for Applied Research in Artificial Intelligence (NCARAI) of the Naval Research Laboratory (NRL).

[Pereira and Cardoso, 2001] Pereira, F. C. and Cardoso, A. (2001). Knowledge Integration with Conceptual Blending. In *12th Irish Conference on Artificial Intelligence & Cognitive Science (AICS 2001)*, Ireland.

[Perkins, 1995] Perkins, D. (1995). An unfair review of Margaret Boden's The Creative Mind from the perspective of creative systems. *Artificial Intelligence*, 79(1):97–109.

[Plate, 1994] Plate, T. (1994). *Distributed Representations and Nested Compositional Structure*. Ph. d., University of Toronto.

[Prieto-Diaz, 1991] Prieto-Diaz, R. (1991). Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):88–97.

[Prieto-Diaz, 1993] Prieto-Diaz, R. (1993). Status Report: Software Reusability. *IEEE Software*, 3(May).

[Prieto-Diaz and Freeman, 1987] Prieto-Diaz, R. and Freeman, P. (1987). Classifying Software for Reusability. *IEEE Software*, 4(January).

[Prieto-Diaz and Jones, 1988] Prieto-Diaz, R. and Jones, G. (1988). Breathing New Life into Old Software. In Tracz, W., editor, *Software Reuse: Emerging Technology*, Washigton, USA. Computer Society Press.

[Pu and Purvis, 1997] Pu, P. and Purvis, L. (1997). Formalizing the Adaptation Process for Case-Based Design. In Maher, M. L. and Pu, P., editors, *Issues and Applications of Case-Based Reasoning in Design*, pag. 221–260, Mahwah, NJ. Lawrence Erlbaum Associates.

[Purvis and Pu, 1995] Purvis, L. and Pu, P. (1995). Adaptation using constraint satisfaction techniques. In Veloso, M. and Aamodt, A., editors, *Proceedings of the 1st International Conference on Case-Based Reasoning*, volume 1010 of *LNAI*, pag. 289–300, Berlin. Springer Verlag.

[Purvis and Pu, 1998] Purvis, L. and Pu, P. (1998). COMPOSER: A Case-Based Reasoning System for Engineering Design. *Robotica*, 16(3):285–295.

[Qian and Gero, 1996] Qian, L. and Gero, J. S. (1996). Funtion-Behaviour-Structure Paths and Their Role in Analogy-Based Design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 10:289–312.

[Racine and Yang, 1997] Racine, K. and Yang, Q. (1997). Maintaining Unstructured Case Bases. In Leake, D. B. and Plaza, E., editors, *Proceedings of the 2nd International Conference on Case-Based Reasoning (ICCBR-97)*, volume 1266 of *LNAI*, pag. 553–564, Berlin. Springer.

[Ram et al., 1995] Ram, A., Wills, L., Domeshek, E., Nersessian, N., and Kolodner, J. (1995). Understanding the creative mind: A review of Margaret Boden's The Creative Mind. *Artificial Intelligence*, 79(1):111–128.

[Reich, 1991] Reich, Y. (1991). Design Knowledge Acquisition: Task Analysis and a Partial Implementation. *Knowledge Acquisition: An International Journal of Knowledge Acquisition for Knowledge-Based Systems*, 3(3):234–254.

[Resnik, 1995a] Resnik, P. (1995a). Disambiguating Noun Groupings with Respect to Wordnet Senses. In Yarovsky, D. and Church, K., editors, *Proceedings of the Third Workshop on Very Large Corpora*, pag. 54–68, Somerset, New Jersey. Association for Computational Linguistics.

[Resnik, 1995b] Resnik, P. (1995b). Using Information Content to Evaluate Semantic Similarity in a Taxonomy. In Mellish, C. S., editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pag. 448–453, San Mateo. Morgan Kaufmann.

[Rigau et al., 1997] Rigau, G., Atserias, J., and Agirre, E. (1997). Combining Unsupervised Lexical Knowledge Methods for Word Sense Disambiguation. In Cohen, P. R. and Wahlster, W., editors, *Proceedings of the Thirty-Fifth Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pag. 48–55, Somerset, New Jersey. Association for Computational Linguistics.

[Rijsbergen, 1979] Rijsbergen, C. J. V. (1979). *Information Retrieval, 2nd edition.* Butterworths, London.

[Rosenberg and Hyatt, 1996] Rosenberg, L. H. and Hyatt, L. E. (1996). Developing a Successful Metrics Programme. In *ESA 1996 Product Assurance Symposium and Software Product Assurance Workshop*, pag. 213–216, ESTEC, Noordwijk, The Netherlands. European Space Agency.

[Rumbaugh et al., 1998] Rumbaugh, J., Jacobson, I., and Booch, G. (1998). *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, MA.

[Schank and Foster, 1995] Schank, R. and Foster, D. (1995). The engineering of creativity: A review of Boden's Creative Mind. *Artificial Intelligence*, 79(1):129–143.

[Simina and Kolodner, 1997] Simina, M. and Kolodner, J. (1997). Creative Design: Reasoning and Understanding. In *International Conference on Case-Based Reasoning (ICCBR 97)*, Providence - Rhode Island, USA. Springer-Verlag.

[Smeaton and Quigley, 1996] Smeaton, A. F. and Quigley, I. (1996). Experiments on Using Semantic Distances Between Words in Image Caption Retrieval. In *19th International Conference on Research and Development in Information Retrieval*, pag. 174–180, Zurich, Switzerland.

[Smith et al., 1995] Smith, I., Lottaz, C., and Faltings, B. (1995). Spatial composition using cases : IDIOM. In Veloso, M. and Aamodt, A., editors, *Proceedings of the 1st International Conference on Case-Based Reasoning*, volume 1010 of *LNAI*, pag. 88–97, Berlin. Springer Verlag.

[Smyth, 1996] Smyth, B. (1996). Case Adaptation & Reuse in Déjà Vu. In *Workshop on Adaptation in Case-Based Reasoning of the 12th European Conference on Artificial Intelligence (ECAI'96)*, Budapest, Hungary.

[Smyth and Cunningham, 1992] Smyth, B. and Cunningham, P. (1992). Deja Vu: A Hierarchical Case-Based Reasoning System for Software Design. In Neumann, B., editor, *10th European Conference on Artificial Intelligence (ECAI'92)*, Vienna, Austria. John Wiley and Sons.

[Smyth and Keane, 1995a] Smyth, B. and Keane, M. (1995a). Experiments on Adaptation-Guided Retrieval in Case-Based Reasoning. In *International Conference on Case-Based Reasoning (ICCBR'95)*, pag. 313–324, Sesimbra, Portugal. Springer-Verlag.

[Smyth and Keane, 1995b] Smyth, B. and Keane, M. T. (1995b). Remembering To Forget: A Competence-Preserving Case Deletion Policy for Case-Based Reasoning Systems. In Mellish, C. S., editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 95)*, pag. 377–383, San Mateo. Morgan Kaufmann.

[Smyth and McKenna, 1998] Smyth, B. and McKenna, E. (1998). Modelling the Competence of Case-Bases. In Smyth, B. and Cunningham, P., editors, *Proceedings of the 4th European Workshop on Advances in Case-Based Reasoning (EWCBR-98)*, volume 1488 of *LNAI*, pag. 208–220, Berlin. Springer.

[Smyth and McKenna, 1999] Smyth, B. and McKenna, E. (1999). Building Compact Competent Case-Bases. In Althoff, K.-D., Bergmann, R., and Branting, L. K., editors, *Proceedings of the 3rd International Conference on Case-Based Reasoning Research and Development (ICCBR-99)*, volume 1650 of *LNAI*, pag. 329–342, Berlin. Springer.

[Spanoudakis and Constantopoulos, 1994] Spanoudakis, G. and Constantopoulos, P. (1994). Similarity for Analogical Software Reuse: A Computational Model. In Cohn, A., editor, *11th European Conference on Artificial Intelligence*, pag. 18–22, Amesterdam, The Netherlands. John Wiley & Sons.

[Sycara and Navinchandra, 1991] Sycara, K. and Navinchandra, D. (1991). Influences: A Thematic Abstraction for Creative Use of Multiple Cases. In *First European Workshop on Case-Based Reasoning*.

[Sycara and Navinchandra, 1993] Sycara, K. and Navinchandra, D. (1993). Case Representation and Indexing for Innovative Design Reuse. In *Workshop of the 13th International Joint Conference on Artificial Intelligence*, France.

[Tautz and Althoff, 1997] Tautz, C. and Althoff, K.-D. (1997). Using Case-Based Reasoning for Reusing Software Knowledge. In Leake, D. and Plaza, E., editors, *International Conference on Case-Based Reasoning (ICCBR'97)*, pag. 156–165, Providence, RI, USA. Springer-Verlag.

[Tessem et al., 1994] Tessem, B., Bjornestad, S., Tornes, K. M., and Steine-Eriksen, G. (1994). ROSA = Reuse of Object-oriented Specifications through Analogy: A Project Framework. IFI Report 16, Department of Information Science, University of Bergen.

[Thagard et al., 1990] Thagard, P., Holyoak, K. J., Nelson, G., and Gochfield, D. (1990). Analog Retrieval by Constraint Satisfaction. *Artificial Intelligent*, 46:259–310.

[Tokuda and Batory, 1995] Tokuda, L. and Batory, D. (1995). Automated Software Evolution via Design Patterns. In *3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico.

[Tong and Sriram, 1992] Tong, C. and Sriram, D. (1992). *Artificial Intelligence in Engineering Design*, volume I. Academic Press.

[Turner, 1995] Turner, S. (1995). Book review of The Creative Mind: Myths and Mechanisms (Margaret Boden). *Artificial Intelligence*, 79(1):145–159.

[Voss, 1996] Voss, A. (1996). Towards a Methodology for Case Adaptation. In *ECAI '96; 12th European Conference on Artificial Intelligence*, pag. 147–154, Chichester - New York - Brisbane. Wiley.

[Voss, 1997] Voss, A. (1997). Case Design Specialists in FABEL. In Maher, M. L. and Pu, P., editors, *Issues and Applications of Case-Based Reasoning in Design*, pag. 301–336, Mahwah, NJ. Lawrence Erlbaum Associates.

[Voss and Coulon, 1996] Voss, A. and Coulon, C.-H. (1996). Structural adaptation with TOPO. In Voss, A., editor, *ECAI 96, Workshop 6, Adaptation in Case-Based Reasoning*, pag. 52–54. ECAI 96.

[Voss et al., 1994] Voss, A., Coulon, C.-H., Grather, W., Linowski, B., Schaaf, J., Bartsch-Sporl, B., Borner, K., Tammer, E. C., Durschke, H., and Knauff, M. (1994). Retrieval of Similar Layouts - About a very Hybrid Approach in FABEL. In Gero, J. S. and Sudweeks, F., editors, *Artificial Intelligence in Design, AID'94*, pag. 625–640, Lausanne, Switzerland. Kluwer Academic Publishers, Netherlands.

[Wiggins, 2001] Wiggins, G. (2001). Towards a more precise characterisation of creativity in AI. In *International Conference on Case-Based Reasoning (ICCBR'01)*

Workshop on Creative Systems: Approaches to Creativity in Artificial Intelligence and Cognitive Science, Vancouver, Canada. Technical Report of the Navy Center for Applied Research in Artificial Intelligence (NCARAI) of the Naval Research Laboratory (NRL).

[Wilks and Stevenson, 1996] Wilks, Y. and Stevenson, M. (1996). The Grammar of Sense: Is word-sense tagging much more than part-of-speech tagging? Technical Report CS-96-05, Department of Computer Science, University of Sheffield, Sheffield, UK.

[Yarowsky, 1992] Yarowsky, D. (1992). Word-Sense Disambiguation using Statistical Models of Roget's Categories Trained on Large Corpora. In *Proceedings of COLING-92*, pag. 454–460, Nantes, France.

[Zhao, 1991] Zhao, F. (1991). *A Knowledge-Based Representation for Creative Design*. Ph. d., Carnegie Mellon University, Department of Civil Engineering.

[Zhu and Yang, 1999] Zhu, J. and Yang, Q. (1999). Remembering to Add: Competence-preserving Case-Addition Policies for Case Base Maintenance. In Thomas, D., editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol1)*, pag. 234–241, S.F. Morgan Kaufmann Publishers.